

Program Development in Java

Abstraction, Specification, and Object-Oriented Design

Scritto da **Barbara Liskov**

Indice

1. Introduzione

1. Decomposizione e Astrazione
2. Astrazione
 1. Astrazione per parametrizzazione
 2. Astrazione per specificazione
 3. Tipi di astrazioni

2. Capire gli oggetti in Java

1. Struttura del programma
2. Pacchetti
3. Oggetti e variabili
 1. Mutabilità
 2. Semantica delle chiamate di metodo
4. Tipo Controllo
 1. Gerarchia dei tipi
 2. Conversione e sovraccarichi
5. Dispatching
6. Tipi
 1. Tipi di oggetti primitivi
 2. Vettori
7. Stream Input/Output

3. Astrazione procedurale

1. I vantaggi dell'astrazione
2. Specifiche
3. Specifiche delle astrazioni procedurali
4. Implementazione delle procedure
5. Progettare astrazioni procedurali

4. Eccezioni

1. Specifiche
2. Il meccanismo delle eccezioni di Java
 1. Tipi eccezione

2. Definizione dei tipi eccezione
 3. Lancio di eccezioni
 4. Gestione delle eccezioni
 5. Gestire le eccezioni non controllate
3. Programmazione con le eccezione
 1. Riflessione e mascheramento
 4. Problemi di progettazione
 1. Quando usare le eccezioni?
 2. Eccezioni controllate o *non* controllate?
 5. Programmazione difensiva

5. Astrazione di dati

1. Specifiche per le astrazioni di dati
 1. Specifiche di IntSet
 2. L'astrazione Poly
2. Utilizzo delle astrazioni di dati
3. Implementazione delle astrazioni di dati
 1. Implementazione delle astrazioni di dati in Java
 2. Implementazione di IntSet
 3. Implementazione di Poly
 4. Records
4. Metodi aggiuntivi
 1. Metodi aggiuntivi: `equals`, `hashCode` e `similar`
 2. Metodi aggiuntivi: `clone`
 3. Metodi aggiuntivi: `toString`
5. Ausili per la comprensione delle implementazioni
 1. La funzione di astrazione
 2. L'invariante di rappresentazione
 3. Implementazione della AF e dell'IR
 4. Discussione
6. Proprietà delle implementazioni dell'astrazione di dati
 1. Effetti collaterali benevoli
 2. Esporre il rappresentante
7. Ragionare sulle astrazioni di dati
 1. Preservare l'invariante di rappresentazione
 2. Ragionare sulle operazioni
 3. Ragionare a livello astratto
8. Problemi di progettazione

1. Mutabilità
2. Categorie di operazioni
3. Adeguatezza

9. Località e modificabilità

6. Astrazione dell'iterazione

1. Iterazione in Java
2. Specificare gli iteratori
3. Utilizzo degli iteratori
4. Implementazione degli iteratori
5. Invarianti di rep e funzioni di astrazione per i generatori
6. Elenchi ordinati
7. Problemi di progettazione

7. Gerarchia dei tipi

1. Assegnamento e Dispatching
 1. Assegnamento
 2. Dispatching
2. Definizione di una gerarchia di tipi
3. Definire le gerarchie in Java
4. Un semplice esempio
5. Tipi di eccezione
6. Classi astratte
7. Interfacce
8. Implementazioni multiple
 1. Liste
 2. Polinomi
9. Il significato dei sottotipi
 1. La regola dei metodi
 2. La regola delle proprietà
 3. Uguaglianza
10. Discussione sulla gerarchia di tipi

8. Astrazioni polimorfiche

1. Astrazioni di dati polimorfi
2. Utilizzo delle astrazioni di dati polimorfi
3. L'uguaglianza rivisitata
4. Metodi aggiuntivi
5. Maggiore Flessibilità
6. Procedure polimorfiche

7. Sintesi

1 - Introduzione

1.1 - Decomposizione e Astrazione

Il paradigma di base per affrontare qualsiasi problema di grandi dimensioni è chiaro: dobbiamo "dividere et impera" e il modo esatto in cui scegliamo di dividere il problema è di importanza fondamentale.

Il nostro obiettivo nella scomposizione di un programma è **creare moduli** che siano essi stessi *piccoli programmi* che interagiscono tra loro in modi semplici e ben definiti. Se raggiungiamo questo obiettivo, persone diverse saranno in grado di lavorare su moduli diversi in modo indipendente, senza bisogno di comunicare molto tra loro, e tuttavia i moduli funzioneranno insieme. Inoltre, durante la modifica e la manutenzione del programma, sarà possibile modificare alcuni moduli senza influenzare tutti gli altri.

Quando si decompone un problema, lo si fattorizza in sottoproblemi separabili in modo tale che:

- Ogni sottoproblema abbia lo stesso livello di dettaglio.
- Ogni sottoproblema può essere risolto in modo indipendente.
- Le soluzioni dei sottoproblemi possono essere combinate per risolvere il problema originale.

Ricorda: i problemi di grandi dimensioni o di scarsa comprensione sono difficili da scomporre correttamente. Il problema più comune è la creazione di singoli componenti che risolvono i sottoproblemi indicati, ma non si combinano per risolvere il problema originale. Questo è uno dei motivi per cui l'integrazione dei sistemi è spesso difficile.

L'**astrazione** è un modo per effettuare la scomposizione in modo produttivo, cambiando il livello di dettaglio da considerare. Quando astraiamo da un problema, accettiamo di ignorare alcuni dettagli nel tentativo di convertire il problema originale in uno più semplice.

Il paradigma dell'astrazione e poi della decomposizione è tipico del processo di progettazione di un programma:

- la decomposizione viene usata per suddividere il software in componenti che possono essere combinati per risolvere il problema originale;
- le astrazioni aiutano a fare una buona scelta dei componenti.

Si alternano i due processi finché non si è ridotto il problema originale a un insieme di problemi che si sa già come risolvere.

1.2 - Astrazione

Il processo di astrazione può essere visto come un'applicazione di una mappatura molti-a-uno: ci permette di dimenticare le informazioni e, di conseguenza, di trattare cose diverse come se fossero

uguali. Lo facciamo nella speranza di semplificare la nostra analisi. È fondamentale ricordare, tuttavia, che la rilevanza dipende spesso dal contesto.

Ricorda: un linguaggio contenente tante astrazioni incorporate potrebbe essere così ingombrante da risultare inutilizzabile.

Un'alternativa preferibile è quella di progettare nel linguaggio meccanismi che permettano ai programmatori di costruire le proprie astrazioni quando ne hanno bisogno. Un meccanismo comune è l'uso delle **procedure**. Separando la definizione e l'invocazione delle procedure, un linguaggio di programmazione rende possibili due importanti metodi di astrazione: l'*astrazione per parametrizzazione* e l'*astrazione per specificazione*.

1.2.1 - Astrazione per parametrizzazione

L'astrazione per parametrizzazione astrae dall'identità dei dati sostituendoli con *parametri*.

Generalizza i moduli in modo che possano essere utilizzati in più situazioni. Ad esempio, nel codice

```
java
int squares (int x,y) {
    return x*x+y*y;
}
```

- “x” e “y” sono chiamati **parametri formali**
- “x * x + y * y” è chiamato **corpo** dell'espressione

Quando una computazione viene invocata, i parametri formali vengono legati agli argomenti (parametri reali) e viene successivamente valutato il corpo, ad esempio:

```
java
u = squares(w, z)
```

Quindi, l'astrazione per parametrizzazione, con l'introduzione di parametri, permette di descrivere un insieme (anche infinito) di computazioni diverse con un singolo programma che le astrae tutte, favorito anche dalla facile realizzazione nei linguaggi di programmazione.

1.2.2 - Astrazione per specificazione

L'astrazione per specificazione astrae dai *dettagli* dell'implementazione (come il modulo è implementato) al *comportamento* da cui gli utenti possono dipendere (ciò che il modulo fa). Ciò avviene associando a ogni procedura una **specifica** dell'effetto che si intende ottenere e considerando il significato di una procedura basato sulla sua specifica piuttosto che sul corpo della procedura stessa.

Si fa uso dell'astrazione per specificazione ogni volta che si associa a una procedura un commento sufficientemente informativo da permettere ad altri di usare quella procedura senza guardare il suo corpo. Un buon modo per scrivere tali commenti è quello di usare coppie di **asserzioni**:

- La “*requires assertion*” (o *precondizione*) di una procedura specifica qualcosa che si presuma sia vero all'ingresso della procedura. In pratica, ciò che viene asserito più spesso è un insieme di condizioni sufficienti a garantire il corretto funzionamento della procedura.
- La “*effects assertion*” (o *post-condizione*) specifica qualcosa che si suppone sia vero al completamento di ogni invocazione della procedura per la quale la precondizione è stata soddisfatta.

```
java
float sqrt (float coef) {
    // REQUIRES: coef > 0
    // EFFECTS: Returns an approximation to the square root of coef
    float ans = coef/2.0;
    int i = 1;
    while (i < 7) {
        ans = ans - ((ans * ans - coef)/(2.0*ans));
    }
    return ans;
}
```

Quando si utilizza una specifica per ragionare sul significato di una chiamata di una procedura, si seguono due regole distinte:

1. Dopo l'esecuzione della procedura, possiamo assumere che la post-condizione sia valida, a condizione che la precondizione sia valida al momento della chiamata.
2. Possiamo assumere *solo* le proprietà che possono essere dedotte dalla post-condizione.

Le due regole rispecchiano i due vantaggi dell'astrazione tramite specifica. La prima afferma che gli utenti della procedura non devono preoccuparsi di esaminare il corpo della procedura per poterla utilizzare. La seconda regola chiarisce che stiamo effettivamente astraendo dal corpo della procedura, cioè omettendo alcune informazioni presumibilmente irrilevanti. Quindi, gli utenti della procedura faranno affidamento al comportamento di essa e non alla sua implementazione.

1.2.3 - Tipi di astrazioni

L'astrazione per parametrizzazione e per specificazione sono metodi per la costruzione di programmi. Ci permettono di definire tre diversi tipi di astrazione: *astrazione procedurale*, *astrazione dei dati* e *astrazione dell'iterazione*. Queste tre tipologie incorporeranno entrambi i metodi di astrazione al loro interno:

- **Astrazione procedurale:** permette di estendere la macchina virtuale definita da un linguaggio di programmazione aggiungendo una nuova operazione. Questo tipo di estensione è molto utile quando abbiamo a che fare con problemi che sono convenientemente scomponibile in unità funzionali indipendenti.
- **Astrazione di dati:** consiste in un insieme di oggetti e in un insieme di operazioni che caratterizzano il comportamento degli oggetti. Il comportamento degli oggetti di dati è espresso in termini di un insieme di operazioni significative per tali oggetti, tra cui “creare oggetti”, “ottenere informazioni da essi” e “modificarli”. Un esempio possono essere le pile, le quali operazioni più significative sono `push` e `pop`.
- **Astrazione dell'iterazione:** consente di iterare sugli elementi di un insieme senza rivelare i dettagli di come gli elementi sono stati ottenuti.

Infine, a volte, astraiamo gruppi di astrazioni di dati in *famiglie di tipo*. Tutti i membri della famiglia hanno operazioni in comune; queste operazioni comuni sono definite nel *supertipo*, ossia il tipo che è l'antenato di tutti gli altri, che sono i suoi *sottotipi*. Tutta questa struttura prende il nome di **gerarchia di tipo**, che permette quindi di astrarre dai singoli tipi di dati alle famiglie di tipi correlati.

2 - Capire gli oggetti in Java

Java è un linguaggio orientato agli oggetti. Ciò significa che la maggior parte dei dati manipolati dai programmi sono contenuti in *oggetti*. Gli oggetti contengono sia lo stato che le operazioni; le operazioni sono chiamate *metodi*. I programmi interagiscono con gli oggetti invocando i loro metodi. I metodi forniscono accesso allo stato, consentendo al codice di osservare lo stato corrente di un oggetto o di modificarlo.

2.1 - Struttura del programma

I programmi Java sono costituiti da *classi* e *interfacce*. **Classi** sono utilizzate in due modi diversi: per definire collezioni di procedure e per definire nuovi tipi di dati. Anche le **interfacce** vengono utilizzate per definire nuovi tipi di dati.

La maggior parte del contenuto delle classi e delle interfacce consiste nella definizione di *metodi*:

- Una classe che definisce un gruppo di procedure fornisce un metodo per ogni procedura; ad esempio, una classe che fornisce procedure che manipolano array di interi può contenere un metodo per ordinare un array e un altro metodo per cercare un array che abbia una corrispondenza con un particolare numero intero.
- Una classe o un'interfaccia che definisce un tipo di dati fornisce metodi per le operazioni associate agli oggetti di quel tipo. Ad esempio, nel caso di un `MultiSet`, potrebbe esserci un metodo `insert` per aggiungere un intero nel `MultiSet` e un metodo `numberOf` per determinare quante volte un dato intero compare nel `MultiSet`.

Un metodo accetta zero o più argomenti e restituisce un singolo risultato, come indicato dalla sua intestazione. Gli argomenti vengono passati come *parametri formali* o *formali* della chiamata. Poiché Java richiede che ogni metodo abbia un risultato, quando non c'è un risultato si usa un particolare tipo di ritorno, ossia `void`.

2.2 - Pacchetti

Le classi e le interfacce sono raggruppate in **pacchetti**. I pacchetti hanno due scopi/utilizzi:

- **encapsulation**, ossia forniscono un modo per condividere informazioni all'interno del pacchetto, impedendone l'uso all'esterno. Ciò è favorita da una *visibilità* (`public`, `private`, `protected`) dichiarata in ogni classe, interfaccia e dalle loro dichiarazioni.
- **denominazione**: ogni pacchetto ha un nome gerarchico che lo distingue da tutti gli altri pacchetti. Le classi e le interfacce all'interno del pacchetto hanno nomi relativi al nome del pacchetto. Ciò significa che non ci sono conflitti di nome tra classi e interfacce definite in pacchetti diversi. Il codice di un pacchetto può fare riferimento ad altre classi e interfacce del proprio pacchetto utilizzando il nome della classe o dell'interfaccia. Le definizioni in altri pacchetti possono essere indicate usando l'intero nome (es. `mathRoutines.Num`). È anche

possibile usare nomi brevi per riferirsi a definizioni in altri pacchetti, usando l'istruzione `import` per importare tutte le definizioni pubbliche da un pacchetto o per importare definizioni pubbliche specifiche da un pacchetto.

Un problema dei nomi brevi è la possibilità di **conflitti** tra i nomi. Si può usare un nome completamente qualificato per ciascuna di esse oppure importare una delle classi e usare un nome lungo per l'altra.

A volte c'è un conflitto tra incapsulamento e denominazione. È conveniente raggruppare molte definizioni nello stesso pacchetto, perché in questo modo il codice esterno al pacchetto può accedere a tutte le definizioni importando l'intero pacchetto. Ma questo tipo di raggruppamento può essere sbagliato dal punto di vista dell'incapsulamento, perché il codice all'interno di un pacchetto può talvolta accedere alle informazioni interne di altre definizioni all'interno dello stesso pacchetto. In generale, un simile conflitto dovrebbe essere risolto a favore dell'incapsulamento.

2.3 - Oggetti e variabili

Tutti i dati sono accessibili tramite *variabili*. Le *variabili locali*, come quelle dichiarate all'interno dei metodi, risiedono nella *stack* di runtime: lo spazio viene allocato per esse quando il metodo viene chiamato e deallocato quando il metodo ritorna.

Ogni variabile ha una dichiarazione che ne indica il suo *tipo*. I *tipi primitivi*, come `int` (*integers*), `boolean` e `char` (*characters*), definiscono insiemi di valori, come `3`, `false` o `c`. Tutti gli altri tipi definiscono insiemi di oggetti.

Alcuni tipi di oggetti vengono forniti in pacchetti definiti da altri. Uno di questi è `java.lang`, che fornisce una serie di tipi utili come `String`; i tipi definiti da questo pacchetto possono essere utilizzati senza importare il pacchetto, diversamente da altri (quelli che dovremmo definire noi stessi).

Le variabili di tipo primitivo contengono *valori* (es. `int i = 6;`), che memorizza il valore 6 nella variabile `i`.

Le variabili di tutti gli altri tipi, comprese le stringhe e gli array, contengono *referimenti* a *oggetti* che risiedono sull'*heap*. Gli oggetti vengono creati sull'heap utilizzando l'operatore `new`:

```
int[] a = new int[3];
```

In particolare, `new` alloca sull'heap lo spazio per un nuovo oggetto array di numeri interi, con spazio per tre numeri interi. Infine, un riferimento al nuovo oggetto viene memorizzato nella variabile `a`.

Le variabili locali possono essere inizializzate quando vengono dichiarate. Esse *devono essere inizializzate* prima del loro primo utilizzo, altrimenti la compilazione fallirà:

```
int i = 6;
```

```
int j; // non inizializzato
int[] a = {1,3,5,7,9}; // crea un array di 5 elementi
int[] b = new int[3];
String s = "abcdef"; // crea una nuova stringa
String t = null;
```

Nota bene: la variabile `t` è stata inizializzata a `null`. Questo valore speciale fornisce un modo per inizializzare una variabile che alla fine farà riferimento a un oggetto.

L'assegnazione a `b` mostra il modo consueto di creare un nuovo oggetto, chiamando l'operatore *built-in* `new`. Questo operatore crea un oggetto della classe indicata sull'heap e poi lo inizializza eseguendo un tipo speciale di metodo, chiamato *congiugatore*, per quella classe.

Ogni oggetto ha un'identità distinta da quella di ogni altro oggetto. In altre parole, quando un oggetto viene creato con una chiamata a `new`, o attraverso l'uso di forme speciali come `"abcdef"` per le stringhe e `{1,3,5,7,9}` per gli array, ciò che si ottiene è un oggetto distinto da qualsiasi altro oggetto esistente.

Un incarico `v = e` copia il valore ottenuto dalla valutazione dell'espressione `e` nella variabile `v`:

```
i = j;
b = a;
t = s;
```

Nota bene: le variabili stringa e array puntano allo stesso oggetto. Pertanto, l'assegnazione che coinvolge i riferimenti fa sì che le variabili *condividano* gli oggetti.

L'operatore `==` può essere utilizzato per:

- Per i tipi primitivi, determinare se due variabili contengono lo stesso valore.
- Determinare se una variabile che potrebbe riferirsi a un oggetto contiene invece `null`.
- Determinare se due variabili si riferiscono allo stesso oggetto.

Gli oggetti nell'heap continuano a esistere finché sono raggiungibili da qualche variabile dello stack, direttamente o attraverso un percorso di altri oggetti. Quando un oggetto non è più raggiungibile, la sua memoria diventa disponibile per essere recuperata dal *garbage collector*.

2.3.1 - Mutabilità

Gli oggetti possono essere *immutabili* o *mutabili*:

- Oggetto *immutabile*: lo stato di tale oggetto NON cambia mai
- Oggetto *mutabile*: lo stato di tale oggetto cambia

Le stringhe sono immutabili; sebbene le stringhe abbiano l'operatore di concatenazione `+`, esso non modifica nessuno dei suoi argomenti, ma restituisce una nuova stringa il cui stato è la concatenazione

degli stati dei suoi argomenti.

Es:

```
String s = "abcdef"
t = s;
t = t + "g";
```

Adesso `t` fa riferimento a un nuovo oggetto `String` il cui stato è `"abcdefg"` e l'oggetto a cui fa riferimento `s` non viene toccato.

D'altra parte, gli array sono mutabili. L'assegnazione `a[i] = e;` fa cambiare lo stato della matrice `a` sostituendo il suo elemento `i`-esimo con il valore ottenuto valutando l'espressione `e`.

Se un oggetto mutabile è condiviso da due o più variabili, le modifiche effettuate da una di esse saranno visibili quando l'oggetto verrà utilizzato attraverso l'altra variabile.

2.3.2 - Semantica delle chiamate di metodo

Un tentativo di chiamare un metodo `e.m(...)`, valuta prima `e` per ottenere la classe o l'oggetto il cui metodo viene chiamato. Poi vengono valutate le espressioni degli argomenti per ottenere i valori dei *parametri effettivi*; la valutazione avviene da sinistra a destra. Quindi viene creato un *record di attivazione* per la chiamata e viene inserito nello stack; il record di attivazione contiene spazio per i parametri formali del metodo e qualsiasi altra memoria locale richiesta dal metodo. Successivamente, i parametri effettivi vengono assegnati ai formali; questo tipo di passaggio di parametri è chiamato *call by value*. Infine, il controllo viene *trasferito* al metodo chiamato `e.m`.

Nota bene: proprio come nel caso dell'assegnazione alle variabili, se il valore di un parametro effettivo è un riferimento a un oggetto, tale riferimento viene assegnato alla procedura formale. Ciò significa che la procedura chiamata condivide gli oggetti con il suo chiamante. Inoltre, se questi oggetti sono mutabili e la procedura chiamata modifica il loro stato; tali modifiche sono visibili al chiamante quando ritorna.

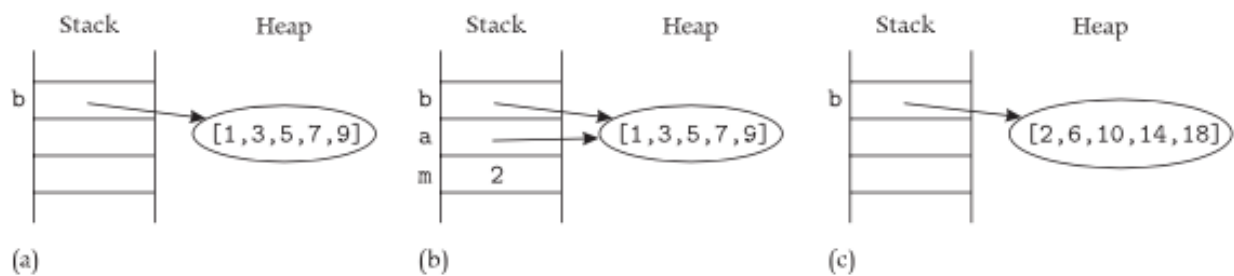
Ad esempio:

```
java
// Dato il main
int[] b = {1, 3, 5, 7, 9};
Arrays.multiples(b,2);

public static void multiples(int[] a, int m) {
    if (a==null) return;
    for (int i = 0; i<a.length;i++) a[i] = a[i]*m;
}
```

La chiamata al metodo avviene in questo modo:

1. Prima della chiamata, assistiamo all'allocazione dell'array `b` nello stack
2. Lo stack, subito dopo la chiamata, contiene il record di attivazione della chiamata: i formali sono stati inizializzati per contenere gli effettivi. Pertanto, il formale `a` di `Arrays.multiples` si riferisce allo stesso array di `b`.
3. Una volta che il metodo ritorna un valore, il record di attivazione creato per la chiamata viene scartato. Tuttavia, l'array degli argomenti è stato modificato e questa modifica è visibile al chiamante attraverso la variabile `b`.



In una chiamata `e.m`, in cui si suppone che `e` valga come oggetto, è possibile che `e` potrebbe invece valutare `null` e quindi non riferirsi ad alcun oggetto. In questo caso, la chiamata non viene effettuata, ma viene sollevata la `NullPointerException`.

2.4 - Tipo Controllo

Java è un linguaggio **fortemente tipizzato**. Il che significa che il compilatore Java controlla il codice per assicurarsi che ogni assegnazione e ogni chiamata sia di tipo corretto. Se viene rilevato un errore di tipo, la compilazione fallisce con un messaggio di errore.

Il controllo dei tipi dipende dal fatto che:

- ogni dichiarazione di variabile fornisce il tipo di variabile
- l'intestazione di ogni metodo e costruttore definisce la sua *signature*: i tipi dei suoi argomenti e dei suoi risultati.

Queste informazioni permettono al compilatore di dedurre un *tipo apparente* per qualsiasi espressione.

Es.

```
java
int y = 7;
int z = 3;
int x = Num.gcd(z, y);
```

Tale deduzione permette di determinare la legalità di un'assegnazione.

I programmi Java legali, cioè quelli accettati dal compilatore, sono garantiti come *type safe*, ossia non

ci possono essere errori di tipo durante l'esecuzione del programma: non è possibile che il programma manipoli dati appartenenti a un tipo come se appartenessero a un tipo diverso. La sicurezza dei tipi si ottiene con tre meccanismi:

1. Controllo dei tipi in fase di compilazione, in quanto Java è un linguaggio fortemente tipizzato.
2. Gestione automatica della memoria
3. Controllo dei limiti degli array; Java controlla tutti gli accessi agli array per verificare che rientrino nei limiti.

2.4.1 - Gerarchia dei tipi

I tipi di Java sono organizzati in una *gerarchia* in cui un tipo può avere un certo numero di *supertipi*. In particolare, diciamo che un tipo è un **sottotipo** (*sottoclasse*) di ciascuno dei suoi **supertipi** (superclasse). La gerarchia dei tipi fornisce un modo per astrarre dalle differenze tra i sottotipi al loro comportamento comune, catturato dal loro supertipo. La relazione di sottotipo è transitiva: se R è un sottotipo di S e S è un sottotipo di T, allora R è un sottotipo di T. La relazione è anche riflessiva: il tipo S è un sottotipo di se stesso.

Se S è un sottotipo di T, i suoi oggetti sono destinati a essere utilizzabili in qualsiasi contesto che preveda l'uso di oggetti appartenenti a T. Perché gli oggetti S siano utilizzabili, devono avere tutti i metodi che hanno gli oggetti T; questo requisito è imposto dal compilatore Java.

Inoltre, tutte le chiamate di metodo devono comportarsi allo stesso modo sugli oggetti S e T; questo requisito *non* è applicato da Java.

Il tipo speciale `Object` si trova in cima alla gerarchia dei tipi in Java; tutti i tipi di oggetti, compresi i tipi `String` e array, sono sottotipi di questo tipo. Ciò significa che tutti gli oggetti hanno determinati metodi, in particolare quelli specificati per `Object`. Ad esempio, i metodi di `Object` includono `equals` e `toString`, con le seguenti intestazioni:

```
boolean equals (Object o)
String toString()
```

Poiché gli oggetti di un sottotipo si comportano come quelli di un supertipo, ha senso permettere che vengano riferiti a una variabile il cui tipo dichiarato è un supertipo. Questo uso è consentito da Java: un'assegnazione `v=e` è legale se il tipo di `e` è un sottotipo del tipo di `v`.

Nota bene: un'implicazione della regola di assegnazione è che il *tipo effettivo* di un oggetto ottenuto valutando un'espressione è un sottotipo del tipo apparente dell'espressione dedotto dal compilatore tramite le dichiarazioni.

Ad esempio:

```
java
String s = "abc";
Object o2 = s;
```

In questo caso, il tipo apparente di `o2` è `Object`, ma il suo tipo effettivo è `String`.

La verifica del tipo viene sempre effettuata utilizzando il tipo apparente, ossia il tipo compreso dal compilatore in base alle informazioni disponibili nelle dichiarazioni. Ciò significa, ad esempio, che qualsiasi chiamata di metodo effettuata utilizzando l'oggetto sarà determinata come legale in base al tipo apparente.

Nell'esempio precedente solo i metodi degli oggetti, come `equals`, possono essere chiamati su `o2`; i metodi delle stringhe, come `length`, non possono essere chiamati.

A volte un programma ha bisogno di determinare il tipo effettivo di un oggetto in fase di esecuzione, ad esempio per poter chiamare un metodo non previsto dal tipo apparente. Questo può essere fatto con il **casting**. L'uso di un cast comporta un controllo in fase di esecuzione; se il controllo ha successo, la computazione indicata è consentita, altrimenti viene sollevata la `ClassCastException`.

Ad esempio:

```
java
if ((String) o2.length()) // legale
s = (String) o2; // legale
```

I cast, in questo caso, controllano se il tipo effettivo di `o2` è il tipo uguale al tipo `String` indicato; questi controlli hanno esito positivo e quindi l'assegnazione nella prima istruzione o la chiamata al metodo nella seconda istruzione sono consentite.

Nota bene: Java garantisce che il tipo apparente di qualsiasi espressione sia un supertipo del suo tipo effettivo.

2.4.2 - Conversione e sovraccarichi

La determinazione della correttezza del tipo non è così semplice come descritto in precedenza, per due motivi:

1. Java consente alcune **conversioni** implicite di un valore di un tipo in un valore di un altro tipo. Le conversioni implicite coinvolgono solo i tipi primitivi, come, ad esempio, da `char` a `int`. In generale, le conversioni comportano un calcolo, producono cioè un nuovo valore (del tipo della variabile) che viene assegnato alla variabile. Dopo aver determinato la conversione necessaria per rendere legale l'assegnazione, il compilatore genera il codice necessario per produrre il nuovo valore. È possibile imparare quali conversioni sono legali e quali calcoli comportano, consultando un testo Java.
2. Java consente l'**overloading**. Ciò significa che possono esistere più definizioni di metodi con lo stesso nome, oltre agli operatori (es. `+` per i tipi `float` e `int`).

Es.

```

java
static int comp(int, long) //def. 1
static int comp(long, int) //def. 2
static int comp(long, long) //def. 3

int x;
long y;
float z;

// Nota bene: in Java, "int" e "float" possono essere allargati in "long"

```

La chiamata `C.comp(x, y)` potrebbe essere indirizzata sia alla prima definizione di `comp` (poiché in questo caso i tipi corrispondono esattamente) sia alla terza definizione di `comp` (allargando `x` a un `long`). La seconda definizione non è possibile, poiché non è possibile allargare un `long` a un `int`.

La regola utilizzata per determinare quale metodo chiamare quando ci sono diverse scelte, come in questo esempio, è scegliere il metodo "**più specifico**".

Un metodo `m1` è più **specifico** di un altro metodo `m2` se qualsiasi chiamata legale di `m1` sarebbe anche una chiamata legale di `m2` se venissero effettuate più conversioni.

Nell'esempio, `C.comp(x, y)` si riferirebbe alla prima definizione di `comp`, perché più specifica della terza.

Se non esiste un metodo più specifico, si verifica un errore in fase di compilazione.

Ad esempio, tutte e tre le definizioni sono possibili corrispondenze per la chiamata `C.comp(x, x)`. Tuttavia, nessuna di queste è la più specifica e quindi la chiamata è illegale. Il programmatore può risolvere l'ambiguità in un caso come questo rendendo esplicita la conversione; ad esempio, `C.comp((long) x, x)` seleziona la seconda definizione.

Infine, le decisioni di sovraccarico tengono conto anche delle assegnazioni da sottotipi a supertipi.

Ad esempio:

```

java
void foo(T a, int x) // def. 1
void foo(S b, long y) // def. 2

```

Allora `C.foo(e, 3)`, dove `S` è un sottotipo di `T` ed `e` è una variabile di tipo `S`, non è legale poiché nessuna delle due definizioni è più specifica.

2.5 - Dispatching

Quando un metodo viene chiamato su un oggetto, è essenziale che la chiamata vada al codice fornito da quell'oggetto per quel metodo, perché solo quel codice può fare la cosa giusta.

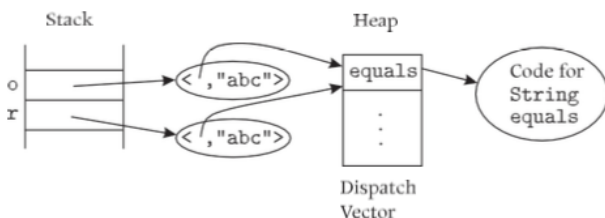
Ad esempio:

```
java
String t = "ab";
Object o = t + "c"; //concatenazione
String r = "abc";
boolean b = o.equals(r);
```

In questo caso si vuole scoprire se il valore di `o` è la stringa `"abc"`. Questo desiderio sarà soddisfatto se la chiamata va al codice di `equals` della stringa, poiché questo confronterà i valori delle due stringhe.

Il problema è che il compilatore non sa necessariamente quale codice chiamare a tempo di compilazione, perché conosce solo il tipo apparente dell'oggetto e non il suo tipo effettivo. Questo è illustrato nell'esempio: il compilatore sa solo che `o` è un oggetto. Se si usasse il tipo apparente per determinare il codice da chiamare, si otterrebbe un risultato errato; per esempio, `b` conterrebbe `false` perché `o` e `r` sono oggetti distinti. Pertanto, abbiamo bisogno di un modo per *collegare* una chiamata di metodo al codice dell'oggetto reale. Ciò richiede un meccanismo di runtime, poiché il compilatore non può capire cosa fare in fase di compilazione.

Il funzionamento è tale:



1. Il vettore di *dispatching* contiene una voce per ogni metodo dell'oggetto
2. Il compilatore genera il codice per accedere alla posizione nel vettore che punta al codice del metodo che viene chiamato e creare un ramo a tale codice

2.6 - Tipi

2.6.1 - Tipi di oggetti primitivi

I tipi primitivi come `int` e `char` non sono sottotipi di `Object` e i loro valori, come `3` e `c`, non possono essere utilizzati in contesti in cui sono richiesti oggetti. Tuttavia, i valori primitivi possono essere utilizzati in contesti che richiedono oggetti, *avvolgendoli* in oggetti. Ogni tipo primitivo ha un tipo di oggetto associato (ad esempio, `Integer` per `int`, `Character` per `char`). Un tale tipo fornisce un costruttore per produrre uno dei suoi oggetti avvolgendo un valore del tipo primitivo associato e un metodo per effettuare la trasformazione inversa:

```
public Integer(int x) // costruttore
public int intValue() // metodo
```

Questi tipi forniscono anche metodi per produrre oggetti dei tipi associati a partire da stringhe:

```
String s = "1024";
int n = Integer.parseInt(s); // n = 1024
```

Se `s` non può essere interpretato come un intero, il metodo lancia `NumberFormatException`.

2.6.2 - Vettori

I vettori sono array estensibili; sono vuoti quando vengono creati per la prima volta e possono crescere e ridursi all'estremo.

Come un array, un vettore contiene elementi numerati da zero fino a uno in meno della sua lunghezza attuale. La lunghezza di un vettore può essere determinata richiamando il suo metodo `size`.

Ogni elemento del vettore ha il tipo apparente `Object`. Ciò significa che i vettori possono essere eterogenei: i diversi elementi di un vettore possono essere oggetti di tipo diverso, sebbene tipicamente il loro utilizzo si limita a elementi dello stesso tipo o di pochi tipi strettamente correlati.

```
Vector v = new Vector(); // crea un vettore nuovo e vuoto
if (v.size() == 0) // true
```

È possibile far crescere un vettore utilizzando il metodo `add` per aggiungere un elemento al suo estremo superiore, ad esempio,

```
v.add("abc");
```

Questo metodo aumenta la dimensione del vettore di 1 e memorizza il suo argomento nella nuova posizione.

È possibile accedere agli elementi del vettore per indici legali con il metodo `get`:

```
String s = (String) v.get(0);
```

Nota bene: nel vettore, `get` restituisce un `Object` e il codice che lo utilizza deve quindi eseguire il cast del risultato nel tipo appropriato.

Se l'indice dato non rientra nei limiti, `get` lancia l'eccezione `IndexOutOfBoundsException`.

Il metodo `set` viene utilizzato per modificare un elemento particolare:

```
v.set(0, "def"); // ora v contiene il singolo elemento "def"
```

Infine è possibile far ridurre il vettore utilizzando il metodo `remove`:

```
v.remove(0);
```

Ricorda: Poiché tutti gli elementi di un vettore devono appartenere a tipi che sono sottotipi di `Object`, i vettori non possono contenere elementi di tipi primitivi come `int` e `char`. Tali valori possono essere memorizzati in un vettore utilizzando i tipi di oggetto associati:

```
v.add(3); // errore in compile time
v.add(new Integer(3)); // legale
```

Per utilizzare un elemento di questo tipo in un secondo momento, deve essere sia lanciato che convertito in un valore:

```
int x = ((Integer) v.get(2)).intValue();
```

2.7 - Stream Input/Output

L'input/output (I/O) avviene tramite flussi di caratteri.

- L'input viene effettuato utilizzando oggetti che appartengono al tipo `Reader` o a uno dei suoi sottotipi. Ad esempio, gli oggetti `BufferedReader` possono essere utilizzati per leggere i caratteri da un flusso, dove il termine *buffer* indica che l'input avviene in pezzi più grandi di singoli caratteri e che i dati vengono conservati in un buffer finché non vengono utilizzati. Il contenuto di un file può essere letto utilizzando il sottotipo `FileReader`:

```
java
FileReader in = new FileReader(filename);
```

- L'output avviene su oggetti di tipo `Writer` o su un sottotipo di questo tipo. Il sottotipo `PrintWriter` può essere utilizzato per stampare valori e oggetti su un dispositivo di output, mentre il sottotipo `FileWriter` può essere utilizzato per inviare l'output a un file.

Java fornisce anche alcuni oggetti predefiniti per eseguire l'I/O standard; questi oggetti sono definiti nella classe `System` del pacchetto `java.lang`:

```
System.in // Standard input al programma
System.out // Standard output dal programma
System.err // Output di errore dal programma
```

Questi oggetti *non sono* oggetti *character stream*. In particolare, `System.in` è un `InputStream`, mentre `System.out` e `System.err` sono `PrintStream`. Tuttavia, questa differenza non deve preoccupare più di tanto, perché gli `InputStream` si comportano come i `Readers` (cioè hanno gli stessi metodi) e gli `OutputStream` sono simili ai `Writers`. Inoltre, è possibile utilizzarli come flussi di caratteri, avvolgendoli; per esempio

```
PrintWriter myOut = new PrintWriter(System.out);
BufferedReader myIn = new BufferedReader(System.in);
```

La maggior parte dei metodi sui flussi effettua un controllo degli errori (ad esempio, per verificare la fine del file quando i dati vengono immessi da un file) e lancia una `IOException` se viene rilevato un errore.

2.8 - Applicazioni Java

Esistono due tipi di applicazioni Java: quelle eseguite dalla riga di comando di un terminale e quelle eseguite interagendo con un'interfaccia utente.

Le applicazioni eseguite dalla riga di comando forniscono un metodo `main`:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Poiché il nome del metodo è `main`, il programma può essere eseguito dalla riga di comando.

3 - Astrazione procedurale

3.1 - I vantaggi dell'astrazione

Un'astrazione è una mappa multi-a-uno. Essa "astrae" dai dettagli "irrilevanti", descrivendo solo quelli rilevanti per il problema in questione. Naturalmente, distinguere ciò che è rilevante da ciò che è irrilevante non è sempre facile.

Nell'*astrazione per parametrizzazione*, si astrae dall'identità dei dati utilizzati. L'astrazione è definita in termini di parametri formali; i dati reali sono legati a questi parametri formali quando l'astrazione viene utilizzata. In questo modo, l'identità dei dati reali è irrilevante, ma la presenza, il numero e i tipi di dati reali sono rilevanti. Una virtù di queste generalizzazioni è che riducono la quantità di codice che deve essere scritto e, quindi, modificato e mantenuto.

Nell'*astrazione per specifica*, ci si concentra sul comportamento da cui l'utente può dipendere e si astrae dai dettagli dell'implementazione di tale comportamento. Pertanto, il comportamento - "cosa" viene fatto - è rilevante, mentre il metodo per realizzare quel comportamento - "come" viene fatto - è irrilevante. Un vantaggio fondamentale dell'astrazione per specificazione è che ci permette di passare a un'altra implementazione senza influenzare il significato di qualsiasi programma che utilizza l'astrazione. L'astrazione tramite specifica fornisce un metodo per ottenere una struttura di programma avente due proprietà vantaggiose:

1. **Localizzazione:** l'implementazione di un'astrazione può essere letta o scritta senza dover esaminare l'implementazione di qualsiasi altra astrazione. Per scrivere un programma che utilizza un'astrazione, il programmatore deve capire solo il suo comportamento, non i dettagli della sua implementazione. La localizzazione è vantaggiosa sia durante la stesura di un programma sia in seguito, quando si vuole capire o ragionare sul suo comportamento, ad esempio all'interno di un progetto di gruppo.
2. **Modificabilità:** se l'implementazione di un'astrazione cambia, ma la sua specifica non cambia, il resto del programma non sarà influenzato dalla modifica. La modificabilità porta a un metodo ragionevole di regolazione delle prestazioni; poiché non è saggio investire sforzi per inventare tecniche che evitino colli di bottiglia inesistenti, un metodo migliore è quello di iniziare con un semplice insieme di astrazioni, far funzionare il sistema per scoprire dove sono i colli di bottiglia e quindi reimplementare le astrazioni che sono colli di bottiglia.

3.2 - Specifiche

Per ottenere i vantaggi discussi, è essenziale che alle astrazioni vengano date definizioni precise. Definiamo le astrazioni per mezzo di **specifiche**, che sono scritte in un *linguaggio di specificazione* che può essere formale o informale. Il vantaggio delle specifiche formali è che hanno un significato preciso, diversamente da quelle informali che, sebbene sono più facili da leggere e scrivere di quelle

formali, è difficile dare loro un significato preciso. Ciononostante, le specifiche informali possono essere molto informative e possono essere scritte in modo tale che i lettori non abbiano difficoltà a comprenderne il significato.

Nel libro, verranno utilizzate le **specifiche informali**.

Una specifica è distinta da qualsiasi implementazione dell'astrazione che definisce

Un linguaggio per le specifiche *non è un* linguaggio di programmazione. Inoltre, le specifiche sono di solito molto diverse dai programmi, perché si concentrano sulla descrizione dell'abitudine piuttosto che sulla sua implementazione. Questo permette di essere molto più brevi e facili da leggere rispetto alla corrispondente implementazione.

3.3 - Specifiche delle astrazioni procedurali

La specifica di una procedura consiste in un'intestazione e in una descrizione degli effetti.

L'*intestazione* fornisce il nome della procedura, il numero, l'ordine e i tipi dei suoi parametri e il tipo del suo risultato; elenca anche le eventuali eccezioni lanciate dalla procedura. Inoltre, è necessario indicare i nomi dei parametri.

Ad esempio, le intestazioni di `removeDups` e `sqrt` sono:

```
java
void removeDups(Vector v);

void sqrt(float x);
```

Le informazioni contenute nell'intestazione sono sintattiche; descrivono la "forma" della procedura. Non viene descritto il significato, dal momento che viene catturato nella parte semantica della specifica.

La parte semantica di una specifica è composta da tre clausole (*requires*, *modifies*, *effects*), mostrate come commenti. Le clausole descrivono una relazione tra gli input e i risultati della procedura. Per la maggior parte delle procedure, gli input sono esattamente i parametri elencati nell'intestazione della procedura. Tuttavia, alcune procedure hanno degli input *impliciti* aggiuntivi (es. un file e `System.out`).

- **Clausola Requires:** indica i vincoli sotto i quali è definita l'astrazione. La clausola *requires* è necessaria se la procedura è **parziale**, cioè se il suo comportamento non è definito per alcuni ingressi. Se la procedura è *totale*, cioè se il suo comportamento è definito per tutti input, la clausola *requires* può essere omessa. In questo caso, le uniche restrizioni su una chiamata legale sono quelle implicite nell'intestazione, cioè il numero e i tipi di argomenti.

- **Clausola Modifies**: elenca i nomi di tutti gli ingressi (compresi quelli impliciti) che vengono modificati dalla procedura. Se alcuni ingressi vengono modificati, si dice che la procedura ha un *side effect*. La clausola **Modifies** può essere omessa quando nessun ingresso viene modificato.
- **Clausola Effects**: descrive il comportamento della procedura per tutti gli ingressi non esclusi dalla clausola **Requires**. Deve definire quali output vengono prodotti e anche quali modifiche vengono apportate agli input elencati nella clausola **Modifies**. La clausola **Effects** è scritta sotto l'ipotesi che la clausola **Requires** sia soddisfatta e non dice nulla sul comportamento della procedura quando la clausola **Requires** non è soddisfatta.

In Java, le procedure autonome sono definite come *metodi statici* di classi. Per utilizzare un metodo di questo tipo, è necessario conoscere la sua classe. Pertanto, dobbiamo includere questa informazione nella specifica, aggiungendo informazioni sulla classe, come il nome e una breve descrizione del suo scopo.

Inoltre, la specifica indica la visibilità della classe e di ogni procedura autonoma.

Es.

```
java
public class Arrays {
    // OVERVIEW: This class provides a number of standalone procedures
    // that are useful for manipulating arrays of ints.

    public static int search(int[] a, int x)
    // EFFECTS: If x is in a, returns an index where x is stored;
    // otherwise, returns -1.

    public static int searchSorted(int[] a, int x)
    // REQUIRES: a is sorted in ascending order
    // EFFECTS If x is in a, returns an index where x is stored;
    // otherwise, returns -1.

    public static void sort(int[] a)
    // MODIFIES: a
    // EFFECTS: Rearranges the elements of a into ascending order
    // e.g., if a=[3,1,6,1] before the call, on return a=[1,1,3,6].
}
```

Nella specifica, possiamo vedere che:

- **search** e **searchSorted** non modificano i loro input, ma **sort** modifica il suo input, come indicato nella clausola *modifies*.
- Le procedure **sort** e **search** sono *totali*, poiché le loro specifiche non contengono una clausola *requires*. **searchSorted**, invece, è *parziale*.

- La clausola degli effetti non indica cosa fa `searchSorted` se l'argomento non soddisfa questo vincolo.

Nota bene: Poiché la classe e i metodi sono `public`, i metodi possono essere utilizzati da codice esterno al pacchetto che contiene la definizione della classe.

Quando una procedura modifica lo stato di un input, la specifica deve mettere in relazione lo stato dell'oggetto al momento del ritorno con quello al momento della chiamata. La scrittura di tali specifiche può essere semplificata da una notazione che identifichi esplicitamente questi diversi stati.

Riprendendo l'esempio precedente, un modo alternativo di scrivere le specifiche di `sort` è:

```
java
public static void sort(int[] a)
    // MODIFIES: a
    // EFFECTS: Rearranges the elements of a into ascending order.
    // For example, if a = [3,1,6,1], a_post = [1,1,3,6].
```

A volte una procedura deve produrre un nuovo oggetto.

Ad esempio:

```
java
public static int[] boundArray(int[], int n)
    // EFFECTS: Returns a new array containing the elements of a in the
    // order they appear in a except that any elements of a that are
    // greater than n are replaced by n.
```

Un esempio di specifica di una procedura che ha ingressi impliciti è il seguente:

```
java
public static void copyLine()
    // REQUIRES: System.in contains a line of text
    // MODIFIES: System.in and System.out
    // EFFECTS: Reads a line of text from System.in, advances the cursor
    // in System.in to the end of the line, and writes the line on
    // System.out.
```

Si noti che la specifica descrive ciò che la procedura fa agli input impliciti.

In genere, le specifiche vengono scritte prime di scrivere il codice che le implementa. In questo modo, si dovrebbe fornire alla classe un'implementazione scheletrica, costituita solo dalle intestazioni dei

metodi e dalle specifiche.

3.4 - Implementazione delle procedure

L'implementazione di una procedura deve produrre il comportamento definito dalla sua specifica.

Ad esempio:

```
java
public class Arrays {
    // OVERVIEW: This class provides a number of standalone procedures
    // that are useful for manipulating arrays of ints.

    public static int searchSorted(int[] a, int x) {
        // REQUIRES: a is sorted in ascending order
        // EFFECTS: If x is in a, returns an index where x is stored;
        // otherwise, returns -1. uses linear search
        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; return -1;
    }

    // other static methods go here
}
```

Si noti che l'implementazione di `searchSorted` restituisce `-1` quando viene passato `null` al posto dell'array di argomenti. Questo comportamento è coerente con quanto descritto nelle sue specifiche. Tuttavia, una specifica migliore avrebbe potuto trattare questo caso in modo speciale, indicando che dovrebbe essere lanciata un'eccezione.

Si noti anche che nel codice è stato inserito un commento che spiega l'algoritmo in uso; tale commento non è necessario se l'algoritmo è semplice, ma dovrebbe essere inserito in caso contrario.

Esempio: implementazione del QuickSort.

```
java
public class Arrays {
    // overview: ...

    public static void sort (int[ ] a) {
        // MODIFIES: a
        // EFFECTS: Sorts a[0], ..., a[a.length - 1] into ascending order.
        // If (a == null) return;
```

```

        quickSort(a, 0, a.length-1);
    }

    private static void quickSort(int[ ] a, int low, int high) {
        // REQUIRES: a is not null and 0 <= low & high < a.length
        // MODIFIES: a
        // EFFECTS: Sorts a[low], a[low+1], ..., a[high]
        // into ascending order. if (low >= high) return;
        int mid = partition(a, low, high);
        quickSort(a, low, mid);
        quickSort(a, mid + 1, high);
    }

    private static int partition(int[ ] a, int i, int j) {
        // REQUIRES: a is not null and 0 <= i < j < a.length
        // MODIFIES: a
        // EFFECTS: Reorders the elements in a into two contiguous groups,
        // a[i],...,a[res]and a[res+1],...,a[j], such that each element in
        // the second group is at least as large as each element of the
        // first group. Returns res.
        int x = a[i];
        while (true) {
            while (a[j] > x) j--;
            while (a[i] < x) i++;
            if (i < j) { // need to swap
                int temp = a[i]; a[i] = a[j]; a[j] = temp;
                j--; i++;
            }
            else return j;
        }
    }
}

```

Si noti che le routine `quickSort` e `partition` *non sono* dichiarate pubbliche, ma il loro uso è limitato alla classe `Arrays`. Questo è appropriato perché sono solo routine di aiuto e hanno poca utilità di per sé. Ciononostante, ne abbiamo fornito le specifiche, che sono interessanti per chi è interessato a capire come viene implementato `quickSort`, ma non per chi lo usa.

Esempio: Rimozione di duplicati da un vettore

```

java
public class Vectors {
    // OVERVIEW: Provides useful standalone procedures for manipulating

```

```

// vectors.

public static void removeDups(Vector v) {
    // REQUIRES: All elements of v are not null
    // MODIFIES: v
    // EFFECTS: Removes all duplicate elements from v;
    // uses equals to determine duplicates. The order of remaining
    // elements may change.
    if (v == null) return;
    for (int i = 0; i < v.size(); i++) {
        Object x = v.get(i);
        int j = i + 1;
        // remove all dups of x from the rest of v
        while (j < v.size())
            if (!x.equals(v.get(j))) j++;
        else { v.set(j, v.lastElement());
              v.remove(v.size() - 1); }
    }
}

```

Si noti che le specifiche spiegano il significato di "duplicato": esso viene determinato utilizzando il metodo `equals` per confrontare gli elementi del vettore.

3.5 - Progettare astrazioni procedurali

Le procedure vengono introdotte durante la progettazione del programma per abbreviare il codice chiamante e chiarirne la struttura. In questo modo, il codice chiamante diventa più facile da capire e da ragionare. Tuttavia, è possibile introdurre troppe procedure. Ci sono due importanti proprietà che le procedure dovrebbero avere: **minimalità**, **generalità** e **semplicità**

Le procedure dovrebbero essere progettate in modo da essere **minimamente vincolate**; si dovrebbe fare attenzione a limitare i dettagli del comportamento della procedura solo nella misura necessaria. In questo modo, lasciamo maggiore libertà all'implementatore, che di conseguenza può essere in grado di fornire un'implementazione più efficiente. Tuttavia, i dettagli che interessano agli utenti devono essere limitati, altrimenti la procedura non sarà quella necessaria. Un tipo di dettaglio che viene quasi certamente lasciato indefinito è l'algoritmo da utilizzare nell'implementazione. Alcuni dettagli di ciò che fa la procedura possono anche essere lasciati indefiniti, il che porta a una **procedura sottodeterminata**. Ciò significa che per determinati input, invece di un singolo risultato corretto, esiste un insieme di risultati accettabili. Un'implementazione è obbligata a produrre qualche

membro di questo insieme, ma qualsiasi membro va bene.

Le procedure `search` e `searchSorted` sono sottodeterminate perché non è stato specificato esattamente quale indice deve essere restituito se `x` si presenta più di una volta nell'array.

Anche `removeDups` (si veda la Figura 3.7) è sottodeterminato, poiché non preserva necessariamente l'ordine degli elementi nel suo vettore di input.

Quindi, i dettagli che interessano agli utenti devono essere specificati; gli altri possono essere lasciati indefiniti.

Un'astrazione sottodeterminata ha di solito un'**implementazione deterministica**, cioè una che, se chiamata due volte con input identici, si comporta in modo identico sulle due chiamate.

Le implementazioni non deterministiche richiedono l'uso di primitive non deterministiche, dati globali o variabili statiche. Per esempio, un'implementazione potrebbe essere leggere l'orologio di sistema ogni volta che viene chiamata e usare quel valore come modo per produrre un risultato diverso da qualsiasi chiamata precedente.

L'altra proprietà è la **generalità**, che spesso si ottiene utilizzando parametri invece di variabili o assunzioni specifiche.

Per esempio, una procedura che cerca un intero arbitrario in un array, dove l'intero è un argomento della procedura, è più generale di una che funziona solo per un intero specifico. Allo stesso modo, una procedura che funziona su array di qualsiasi dimensione è più generale di una che funziona solo su array di dimensioni fisse.

La generalizzazione di una procedura, tuttavia, è utile solo se ne aumenta l'utilità. Questo è quasi sempre vero quando si eliminano le assunzioni relative alle dimensioni, poiché in questo modo si garantisce che un cambiamento minimo nel contesto di utilizzo richieda una modifica minima, se non nulla, del programma.

Un'altra proprietà importante delle procedure è la **semplicità**. Una procedura deve avere uno scopo ben definito e facilmente spiegabile, indipendente dal contesto di utilizzo. Un buon controllo della semplicità consiste nell'assegnare alla procedura un nome che ne descriva lo scopo. Se è difficile pensare a un nome, potrebbe esserci un problema con la procedura.

Infine, vale la pena di notare che una specifica è l'unica traccia della sua astrazione. Pertanto, è fondamentale che le specifiche siano chiare e precise.

Nota bene: Nella scelta tra una procedura parziale e una totale, dobbiamo fare un compromesso. Da un lato c'è l'efficienza, dall'altro un comportamento sicuro, con meno sorprese potenziali in fase di esecuzione; le procedure parziali non sono sicure come quelle totali, poiché lasciano all'utente il compito di soddisfare i vincoli della clausola *requires*. Quando la clausola *requires* non è soddisfatta, il

comportamento di una procedura parziale è completamente privo di vincoli e ciò può causare il fallimento del programma in uso in modi misteriosi. D'altra parte, le procedure parziali possono essere più efficienti da implementare rispetto a quelle totali. Quindi, cosa è meglio scegliere? La risposta è che dipende dal contesto di utilizzo previsto (utilizzo generale o contesto limitato).

Ricorda: all'implementazione di un'astrazione non è vietato controllare il vincolo dato in una clausola *requires*. Se il controllo indica che la clausola *requires* non è soddisfatta, la procedura può produrre un messaggio di errore, ma un approccio migliore è di solito quello di lanciare un'eccezione. Naturalmente, non ha senso controllare un vincolo quando il controllo è molto costoso.

4 - Eccezioni

Un'astrazione procedurale è una mappatura da argomenti a risultati, con la possibile modifica di alcuni degli argomenti. Gli argomenti sono membri del *dominio della* procedura e i risultati sono membri del suo *intervallo*.

Spesso una procedura ha senso solo per argomenti che rientrano in un sottoinsieme del suo dominio. Un modo per far fronte a questa situazione è quello di usare procedure parziali. Il chiamante di una procedura parziale deve assicurarsi che gli argomenti rientrino nel sottoinsieme consentito del dominio, mentre l'implementatore può ignorare gli argomenti al di fuori di questo sottoinsieme.

Le procedure parziali, tuttavia, sono generalmente una cattiva idea, poiché non c'è alcuna garanzia che i loro argomenti siano nel sottoinsieme consentito e la procedura può quindi essere chiamata con argomenti al di fuori del sottoinsieme. Quando ciò accade, la procedura può andare in loop all'infinito o restituire un risultato errato. Quest'ultimo caso è particolarmente negativo, poiché può portare a un errore sicuro e difficile da rintracciare. Ad esempio, il codice chiamante potrebbe continuare a funzionare utilizzando il risultato errato, danneggiando così database importanti.

Le procedure parziali portano a programmi non robusti. Un **programma robusto** è un programma che continua a comportarsi in modo ragionevole anche in presenza di errori. Idealmente, dovrebbe continuare dopo l'errore fornendo un'approssimazione del suo comportamento in assenza di errore; si dice che un programma di questo tipo fornisce una *degradazione aggraziata*. Nel peggiore dei casi, dovrebbe arrestarsi con un messaggio di errore significativo e senza causare danni ai dati permanenti. Un metodo che migliora la robustezza è l'uso di procedure *totali*, il cui comportamento è definito per tutti gli input del dominio. Se la procedura non è in grado di svolgere la sua funzione "prevista" per alcuni di questi ingressi, almeno può informare il suo chiamante del problema. In questo modo, la situazione viene portata all'attenzione del chiamante, che può intervenire o almeno evitare le conseguenze dannose dell'errore.

Una possibilità per notificare al chiamante l'insorgere di un problema è quella di utilizzare un risultato particolare per trasmettere l'informazione.

Ad esempio: procedura fattoriale che restituisce zero se il suo argomento non è positivo:

```
java
public static int fact(int n)
    // EFFECTS: if n > 0 returns n! else returns 0
```

Questa soluzione non è molto soddisfacente. Poiché la chiamata con argomenti illegali è probabilmente un errore, è più costruttivo trattare questo caso in modo speciale. Inoltre, la restituzione di un risultato speciale può risultare scomoda per il codice chiamante, che deve controllarlo. Inoltre, se ogni valore del tipo di ritorno è un possibile risultato della procedura, la soluzione di restituire un risultato speciale è impossibile, poiché non c'è alcun valore residuo da utilizzare. Inoltre, è auspicabile che l'approccio distingua in qualche modo queste situazioni, in modo

che gli utenti non possano ignorarle per errore. Sarebbe anche opportuno che l'approccio permettesse di separare la gestione di queste situazioni dal normale flusso di controllo del programma. Ciò viene reso grazie all'eccezione.

L'**eccezione** permette a una procedura di terminare *normalmente*, restituendo un risultato, oppure *in modo eccezionale*. Ci possono essere diverse terminazioni eccezionali. In Java, ogni termine eccezionale corrisponde a un diverso *tipo eccezione*. I nomi dei tipi di eccezione sono scelti dal definitore della procedura per trasmettere alcune informazioni sul problema. Ad esempio, il metodo `get` di `Vector` ha `IndexOutOfBoundsException`.

4.1 - Specifiche

Una procedura che può terminare in modo eccezionale è indicata da una **clausola *throws***:

```
throws < list_of_types >
```

Ad esempio, il codice:

```
java
public static int fact (int n) throws NonPositiveException
```

afferma che `fact` può terminare lanciando un'eccezione; in questo caso, lancia un oggetto di tipo `NonPositiveException`.

Una procedura può lanciare più di un tipo di eccezione.

Ad esempio, il codice:

```
java
public static int search(int[] a, int x)
    throws NullPointerException, NotFoundException
    // EFFECTS: If a is null throws NullPointerException; else if x is not
    // in a throws NotFoundException; else returns i such that x = a[i].
```

afferma che la ricerca può lanciare due eccezioni: `NullPointerException` (se `a` è `null`) e `NotFoundException` (se `a` non è `null` e `x` non è in `a`).

Le specifiche di una procedura che lancia eccezioni devono chiarire agli utenti cosa sta succedendo esattamente. Innanzitutto, si richiede che l'intestazione elenchi *tutte le* eccezioni che può lanciare come parte del suo comportamento "ordinario", ad esempio per tutti gli input che soddisfano la clausola *requires*.

In secondo luogo, la clausola degli effetti deve spiegare cosa causa il lancio di ogni eccezione: la sezione degli effetti deve definire cosa fa terminare la procedura con ogni eccezione e quale sia il suo comportamento in ogni caso. Inoltre, se una procedura segnala un'eccezione per un certo

sottoinsieme di argomenti, tale sottoinsieme deve essere escluso dalla clausola *requires*. La terminazione con il lancio di un'eccezione fa parte del comportamento ordinario della procedura.

Ad esempio:

```
java
public static int fact(int n) throws NonPositiveException
    // EFFECTS: If n is non-positive, throws NonPositiveException, else
    // returns the factorial of n.

public static int search(int[] a, int x)
    throws NullPointerException, NotFoundException
    // REQUIRES: a is sorted
    // EFFECTS: If a is null throws NullPointerException; else if x is not
    // in a, throws NotFoundException; else returns i such that a[i] = x.
```

Si noti che la specifica di `search` contiene una clausola *requires* e che, come di consueto, la sua sezione *effects* assume che la clausola *requires* sia soddisfatta.

Quando una procedura ha effetti collaterali, la sua specifica deve chiarire come questi interagiscono con le eccezioni. La sezione *modifies* di una specifica indica che un argomento può essere modificato, ma non dice quando ciò avverrà. Se ci sono eccezioni, è probabile che la modifica avvenga solo per alcune di esse. Ciò che accade esattamente deve essere descritto nella sezione degli effetti.

Ad esempio:

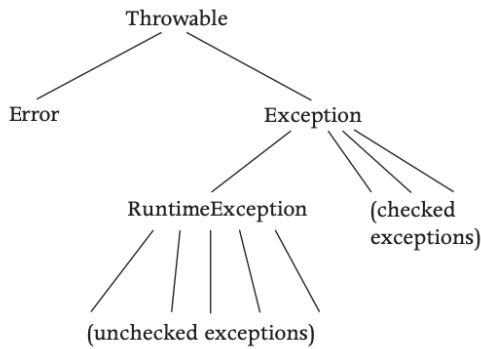
```
java
public static void addMax(Vector v, Integer x)
    throws NullPointerException, NotScmallException
    // REQUIRES: All elements of v are Integers.
    // MODIFIES: v
    // EFFECTS: If v is null throws NullPointerException; if v contains
    // an element larger than x throws NotScmallException; else adds x to
    // v.
```

La specificazione indica che `v` viene modificato solo quando `addMax` ritorna normalmente.

4.2 - Il meccanismo delle eccezioni di Java

4.2.1 - Tipi eccezione

I tipi eccezione sono sottotipi di `Exception` o `RuntimeException`, entrambi sottotipi del tipo `Throwable`. I tipi eccezione seguono una struttura gerarchica:



Esistono due tipi di eccezioni:

- **Eccezioni controllate** (checked): sono sottotipi di **Exception**, ma non di **RuntimeException**. Un esempio è **IOException**.
- **Eccezioni non controllate** (unchecked): sono sottotipi di **RuntimeException** e ricoprono la maggior parte delle eccezioni definite da Java. Alcuni esempi sono **NullPointerException** e **IndexOutOfBoundsException**.

Esistono due differenze nel modo in cui le eccezioni controllate e non controllate possono essere utilizzate in Java:

1. Se una procedura può lanciare un'eccezione controllata, Java richiede che l'eccezione sia elencata nell'intestazione della procedura; in caso contrario, si verificherà un errore di compilazione. Le eccezioni non controllate non devono essere elencate nell'intestazione.
2. Se il codice chiama una procedura che potrebbe lanciare un'eccezione controllata, Java richiede di gestire l'eccezione; in caso contrario, si verificherà un errore di compilazione. Le eccezioni non controllate non devono essere gestite nel codice chiamante.

Tuttavia, la Liskov si discosta dalle regole di Java: viene richiesto che l'intestazione di una procedura elenchi *tutte le* eccezioni che lancia, siano esse controllate o meno. Il motivo è che, dal punto di vista di chi usa la procedura, ogni eccezione che può verificarsi è interessante; non si può capire come usare una procedura senza queste informazioni. Naturalmente si possono ottenere le informazioni dalla clausola degli effetti della specifica, ma includerle nell'intestazione le porta all'attenzione dell'utente in modo molto diretto. Inoltre, fornisce un buon approccio per lo specificatore: elencare tutte le eccezioni nell'intestazione e poi assicurarsi che la clausola degli effetti spieghi ciascuna di esse.

4.2.2 - Definizione dei tipi eccezione

Quando viene definito un nuovo tipo di eccezione, la sua dichiarazione indica se è controllata o meno indicando il suo supertipo: se il supertipo è **Exception** è controllata; mentre se il supertipo è **RuntimeException**, l'eccezione non è controllata.

Una classe che definisce un nuovo tipo di eccezione deve solo definire dei costruttori; ricordiamo che i costruttori sono metodi speciali usati per inizializzare gli oggetti della classe appena creati. La

definizione di un nuovo tipo di eccezione richiede pochissimo lavoro, perché la maggior parte del codice per il nuovo tipo è ereditato dalla classe che implementa il suo supertipo. In particolare, il tipo eccezione prevede due costruttori; il nome del costruttore è sovraccarico.

Ad esempio:

```
java
public class NewKindOfException extends Exception {

    public NewKindOfException() {super();}
    public NewKindOfException(String s) {super(s);}
}
```

fornisce la definizione di un nuovo tipo eccezione controllata, grazie a `extends Exception`.

Nota bene: se si fosse trattato di un tipo eccezione non controllata, nell'intestazione sarebbe contenuto `extends RuntimeException`.

Il primo costruttore inizializza l'oggetto in modo che contenga la stringa vuota.

Ad esempio:

```
java
Exception e2 = new NewKindOfException( );
```

Il secondo costruttore inizializza l'oggetto eccezione in modo che contenga la stringa fornita come argomento, che spiegherà perché l'eccezione è stata lanciata.

Ad esempio:

```
java
Exception e1 = new NewKindOfException("questo è il motivo");
```

fa sì che l'oggetto eccezione e1 contenga la stringa `"questo è il motivo"`.

La stringa, insieme al tipo eccezione, può essere ottenuta chiamando il metodo `toString` sull'oggetto eccezione.

Ad esempio:

```
java
String s = e1.toString( );
```

fa sì che `s` contenga la stringa `"NewKindOfException: questo è il motivo"`.

I tipi di eccezione devono essere definiti in qualche pacchetto. Una possibilità è quella di definirli nello stesso pacchetto che contiene la classe dei metodi che le lanciano. In questo caso, però, sarebbe

necessario un nome più lungo, come `NotFoundFromSearchException`, per evitare conflitti con i tipi di eccezione definiti per altre procedure. Un'alternativa migliore è quindi quella di avere un pacchetto che definisca i tipi di eccezione. In questo modo, lo stesso tipo di eccezione può essere usato in molte routine.

Java non richiede che i tipi di eccezione abbiano la forma `EnameException`. Tuttavia, è un buon stile di programmazione seguire questa convenzione, poiché permette di distinguere facilmente i tipi di eccezione, che devono essere usati solo per lanciare e gestire le eccezioni, dai tipi ordinari.

4.2.3 - Lancio di eccezioni

Una procedura Java può terminare *lanciando* un'eccezione. Lo fa utilizzando l'istruzione **throw**.

Ad esempio:

```
java
if (n <= 0) throw new NonPositiveException("Num.fact");
```

Stiamo lanciando un oggetto di tipo `NonPositiveException`; in realtà costruiamo questo oggetto come parte del lancio, chiamando `new`.

Il problema principale quando si lanciano eccezioni è cosa usare per l'argomento stringa. Per rispondere a questa domanda, è necessario comprendere lo scopo della stringa.

La stringa viene utilizzata principalmente per trasmettere informazioni a una persona quando il programma non è in grado di gestire l'eccezione e quindi si ferma con un messaggio di errore o scrive un messaggio di errore in un registro.

Pertanto, la stringa deve consentire all'utente di scoprire cosa è andato storto. Un buon modo per ottenere questo risultato è che la stringa identifichi **la procedura che ha lanciato l'eccezione**, dato che in generale molte procedure lanciano lo stesso tipo di eccezione. Le informazioni devono permettere di trovare le specifiche di quella procedura. Di solito è sufficiente indicare il nome della classe e del metodo; tuttavia, se il metodo è sovraccarico, è necessario indicare anche i tipi dei suoi argomenti.

4.2.4 - Gestione delle eccezioni

Quando una procedura termina con un'eccezione, l'esecuzione *non* continua subito dopo la chiamata. Il controllo viene invece trasferito a un codice che gestisce l'eccezione.

Il codice gestisce un'eccezione in due modi.

Il primo è quello di gestirla esplicitamente, utilizzando l'istruzione `try`.

Ad esempio:

```
java
```

```
try {x = Num.fact(y); }
catch (NonPositiveException e) {
    // qui dentro è possibile utilizzare e
}
```

Il codice seguente utilizza un'istruzione `try` per gestire la `NonPositiveException` nel caso in cui venga lanciata dalla chiamata di `fact`. Se la chiamata di `fact` lancia `NonPositiveException`, viene eseguita la clausola `catch`: l'oggetto eccezione viene assegnato alla variabile `e`, in modo che questo oggetto possa essere utilizzato durante la gestione dell'eccezione.

Tuttavia, all'istruzione `try` possono essere collegate diverse clausole `catch`, in modo da poter gestire diverse eccezioni. Inoltre, le dichiarazioni `try` possono essere annidate.

Ad esempio:

```
java
try { ...;
    try { x = Arrays.search(v, 7); }
    catch (NullPointerException e) {
        throw new NotFoundException( );
    }
}
catch (NotFoundException b) { ... }
```

In questo caso, la clausola `catch` nell'istruzione `try` esterna gestirà `NotFoundException` se viene lanciata dalla chiamata di `Arrays.search` o dalla clausola `catch` per `NullPointerException`.

Le clausole `catch` non devono identificare il tipo effettivo di un oggetto di eccezione. Al contrario, la clausola può elencare un supertipo del tipo.

Ad esempio:

```
java
try { x = Arrays.search(v, y); }
    catch (Exception e) { s.println(e); return; }
```

La clausola `catch` gestirà sia la `NullPointerException` che la `NotFoundException`. Qui `s` è un `PrintWriter` e `println` usa il metodo `toString` di `e` per ottenere le informazioni da stampare.

Il secondo modo di gestire un'eccezione è quello di propagarla. Ciò si verifica quando una chiamata

all'interno di una procedura `P` segnala un'eccezione che non viene gestita da una clausola `catch` di un'istruzione `try` contenente `P`. In questo caso, Java propaga automaticamente l'eccezione al chiamante di `P`, a condizione che sia vera una delle seguenti condizioni:

- Quel tipo di eccezione o uno dei suoi supertipi è elencato nell'intestazione di `P`
- Il tipo eccezione non è controllato

In caso contrario, si verifica un errore di compilazione. Una procedura dovrebbe sollevare solo le eccezioni elencate nelle sue

specifiche, poiché è su queste che si basa chi scrive il codice che utilizza la procedura.

Fortunatamente, Java non applica questo requisito per le eccezioni non controllate. Pertanto, dovete farlo voi stessi: assicuratevi che qualsiasi eccezione sollevata dal vostro codice, sia per propagazione automatica che per lancio esplicito, sia elencata nell'intestazione della procedura che state implementando (anche se l'eccezione non è verificata) e descritta nelle specifiche di tale procedura.

4.2.5 - Gestire le eccezioni non controllate

Ogni chiamata può potenzialmente lanciare qualsiasi eccezione non controllata. Questo significa che abbiamo un problema nel catturare le eccezioni non controllate, perché è difficile sapere da dove provengono.

Ad esempio:

```
java
try { x = y[n]; i = Arrays.search(z,x); }
catch (IndexOutOfBoundsException e) {
    // gestione IndexOutOfBoundsException dall'accesso dell'array y[n]
}

// il codice continua qua assumendo che il problema sia stato risolto
```

In `IndexOutOfBoundsException`, che è un'eccezione non controllata, non sappiamo dove potrebbe essersi verificata; potrebbe essersi verificata a causa di un errore nell'implementazione di `search`.

L'unico modo per essere certi dell'origine di un'eccezione non controllata è quello di restringere l'ambito dell'istruzione `try`.

Ad esempio, è certo che l'eccezione provenga dall'accesso all'array nel codice seguente:

```
java
try { x = y[n]; }
catch (IndexOutOfBoundsException e) {
    // gestione IndexOutOfBoundsException dall'accesso dell'array y[n]
}
```

```
i = Arrays.search(z,x)
```

4.3 - Programmazione con le eccezione

Quando si implementa una procedura con eccezioni, il compito del programmatore, come sempre, è quello di fornire il comportamento definito dalla specifica. Se questo comportamento include delle eccezioni, il programma deve lanciare le eccezioni corrette al momento giusto, con il significato descritto nella specifica. Per svolgere questo compito, il programma potrebbe dover gestire le eccezioni lanciate dalle procedure che chiama.

Alcune eccezioni sono gestite **in modo specifico**: la clausola catch cerca di rispondere alla situazione specifica che ha dato origine all'eccezione. Altre eccezioni sono gestite **in modo generico**. In questo caso, la clausola catch non cerca di gestire l'eccezione in modo specifico. Al contrario, intraprende un'azione generica. Potrebbe interrompere il programma dopo aver segnalato il problema a un utente, oppure potrebbe "riavviare" il programma tornando a uno stato precedente, senza tentare di risolvere il problema esatto. Ad esempio, un programma di questo tipo potrebbe eseguire una sorta di arresto, seguito da un riavvio pulito. (Lo spegnimento dovrebbe anche essere registrato, in modo che se è dovuto a un errore del programma, l'errore possa essere risolto).

4.3.1 - Riflessione e mascheramento

Ci sono due modi per gestire un'eccezione.

A volte un'eccezione viene **riflessa** a un altro livello, cioè anche il chiamante termina lanciando un'eccezione. Il riflesso di un'eccezione può essere ottenuto tramite una propagazione automatica oppure catturando esplicitamente un'eccezione e poi lanciandola. Il primo caso è più limitato perché viene lanciato lo **stesso oggetto** eccezione. Più comunemente, si vuole lanciare un oggetto diverso, di un tipo di eccezione diverso, perché il significato delle informazioni è cambiato. Un altro punto è che prima di riflettere un'eccezione, il chiamante potrebbe aver bisogno di eseguire alcune

elaborazioni locali per soddisfare le sue specifiche.

Ad esempio:

```
java
public class Arrays {

    public static int min(int[] a)
        throws NullPointerException, EmptyException {
        // EFFECTS: If a is null throws NullPointerException else if a is
        // empty throws EmptyException else returns the minimum value of a
        int m;
        try { m = a[0]; }
        catch (IndexOutOfBoundsException e) {
```

```

        throw new EmptyException("Arrays.min"); }
    for (int i = 1; i < a.length; i++)
        if (a[i] < m) m = a[i];
    return m;
}
}

```

La procedura `min` recupera semplicemente lo zero-esimo elemento dell'array. Se l'argomento dell'array è nullo, la chiamata solleva la `NullPointerException`, che si riflette sul chiamante di `min` e viene propagata automaticamente. Se l'array è vuoto, la chiamata solleva `IndexOutOfBoundsException`. Non avrebbe senso riflettere questa eccezione al chiamante di `min`, poiché vogliamo eccezioni legate all'astrazione di `min` piuttosto che eccezioni che hanno a che fare con il modo in cui `min` è implementato. Invece, `min` lancia `EmptyException`, che è un'eccezione significativa per lui. Si noti che la stringa nell'oggetto eccezione identifica `Arrays.min` come lanciatore.

Una seconda possibilità è che il chiamante *maschi* l'eccezione, ovvero che elimini l'eccezione stessa e poi continui con il normale flusso.

Ad esempio:

```

java

public class Arrays {

    public static boolean sorted(int[] a) throws NullPointerException {
        // EFFECTS: If a is null throws NullPointerException else if a is
        // sorted in ascending order returns true else returns false.
        int prev;
        try { prev = a[0]; }
        catch (IndexOutOfBoundsException e) {return true; }
        for (int i = 1; i < a.length; i++)
            if (prev <= a[i]) prev = a[i]; else return false;
        return true;
    }

}

```

Il codice sta innescando il ciclo; ma in questo caso, se l'array è vuoto, significa semplicemente che è ordinato.

Un punto da notare in entrambi gli esempi è il modo in cui abbiamo usato le eccezioni per controllare il flusso del programma. Questa è una pratica di programmazione perfettamente accettabile: le eccezioni possono essere utilizzate per evitare altre operazioni.

Ad esempio, sia in `min` che in `sorted`, il codice non ha bisogno di controllare esplicitamente la lunghezza dell'array. Tuttavia, a seconda di come è implementato il meccanismo delle eccezioni, potrebbe essere costoso gestire le eccezioni e si dovrebbe soppesare questo costo con il vantaggio di usare le eccezioni per evitare il lavoro extra.

4.4 - Problemi di progettazione

Consideriamo come decidere l'uso delle eccezioni quando si progettano le astrazioni. Le questioni principali sono due: quando usare un'eccezione e se usare un'eccezione controllata o non controllata. Un punto importante è che le **eccezioni non sono sinonimo di errori**. Le eccezioni sono un meccanismo che consente a un metodo di portare all'attenzione del suo chiamante alcune informazioni. Queste informazioni potrebbero non riguardare un errore.

Per esempio, non c'è nulla di errato nel fatto che `search` venga chiamato su un elemento che non è presente nell'array; si tratta invece di una situazione interessante di cui il chiamante deve essere informato. Trasmettiamo questa informazione attraverso un'eccezione perché vogliamo distinguerla dalle altre possibilità.

La classificazione di una possibilità come normale e delle altre come eccezionali è in qualche modo arbitraria.

Inoltre, anche quando un'eccezione è associata a ciò che sembra essere un errore a un livello di astrazione inferiore, la situazione non è necessariamente un errore a un livello superiore.

Per esempio, all'interno del metodo `get` di `Vector`, appare un errore se l'indice dato non è entro i limiti. Tuttavia, dal punto di vista del chiamante di `get`, questa situazione può semplicemente indicare che un ciclo deve terminare. Pertanto, può essere altrettanto "corretto" che una chiamata termini con un'eccezione che terminare normalmente.

Le eccezioni sono semplicemente un mezzo per consentire diversi tipi di comportamento e informare il chiamante sui diversi casi.

Inoltre, non tutti gli errori portano a un'eccezione.

Si consideri un record errato in un file di input di grandi dimensioni, dove è possibile continuare l'elaborazione del file saltando quel record. In questo caso, potrebbe essere opportuno informare una persona (non un programma) dell'errore scrivendo un messaggio di errore su un dispositivo di output

Nota bene: ciò che viene fatto quando si verifica un errore deve essere definito nelle specifiche dell'astrazione, anche quando non viene lanciata alcuna eccezione.

4.4.1 - Quando usare le eccezioni?

Le eccezioni dovrebbero essere usate per **eliminare la maggior parte dei vincoli** elencati nelle clausole *requires*. La clausola *requires* dovrebbe rimanere solo per ragioni di efficienza o se il contesto di utilizzo è così limitato da poter essere sicuri che il vincolo sia soddisfatto.

Ad esempio, `search` potrebbe ancora richiedere l'ordinamento dell'array, poiché può essere implementata in modo molto più efficiente.

Oppure, la procedura `partition` utilizzata nell'ordinamento rapido dovrebbe richiedere che il suo argomento `i` sia inferiore al suo argomento `j`, dato che il contesto di utilizzo è così limitato.

Le eccezioni devono essere utilizzate anche per **evitare di codificare informazioni nei risultati ordinari**.

Ad esempio, invece di restituire `-1` se l'elemento non è presente nell'array, `search` segnala un'eccezione. È meglio trasmettere questa informazione con un'eccezione, poiché il risultato restituito in questo caso non può essere utilizzato come un risultato normale. Utilizzando un'eccezione, è facile distinguere questo risultato da uno normale, evitando così un potenziale errore.

Tuttavia, per le procedure che saranno utilizzate in un contesto limitato (ad esempio, non pubblico), la codifica delle informazioni nei risultati ordinari può essere accettabile.

4.4.2 - Eccezioni controllate o non controllate?

Le **eccezioni controllate** devono essere gestite nel codice chiamante o devono essere elencate nella clausola `throws` dell'intestazione della procedura, altrimenti si verificherà un errore a tempo di compilazione. Questo fornisce una certa **protezione**.

Le **eccezioni non controllate** saranno implicitamente propagate al chiamante, anche se non sono elencate nell'intestazione. Potrebbe sembrare che questo non sia un problema, dato che l'uso del codice può gestire l'eccezione, ad esempio, a livello superiore. Ma il codice non è molto bravo a gestire gli errori dei programmatori, che di solito sono il motivo per cui le eccezioni non controllate si propagano. Inoltre, c'è il rischio che l'eccezione venga catturata per errore. Nel momento in cui l'errore verrà scoperto, potrebbe essere molto difficile da rintracciare.

Perché Java ha eccezioni non controllate quando sono un problema? Il motivo è che anche le eccezioni controllate sono un **problema**: se il vostro codice è certo di non provocare il sollevamento di un'eccezione, dovete comunque gestirla! Questo è il motivo per cui molte eccezioni definite da Java sono di fatto non controllate.

Quindi ci sono buone ragioni da entrambe le parti. Questo significa che c'è un problema di progettazione: quando si definisce un nuovo tipo di eccezione, bisogna pensare bene se deve essere controllata o meno.

La scelta tra eccezioni controllate e non controllate deve basarsi sulle aspettative di utilizzo dell'eccezione. Se si prevede che l'uso del codice eviterà le chiamate che sollevano l'eccezione, l'eccezione dovrebbe essere non controllata.

Questa è la logica alla base di `IndexOutOfBoundsException`: gli array dovrebbero essere usati principalmente in cicli `for` che controllano gli indici e quindi assicurano che tutte le chiamate ai metodi degli array abbiano indici entro i limiti.

Altrimenti, è necessario controllare le eccezioni.

Ad esempio, è probabile che molte chiamate di `search` vengano effettuate senza sapere se l'intero cercato si trova nell'array. In questo caso, sarebbe un errore per il codice chiamante non gestire l'eccezione. Pertanto, il tipo di eccezione deve essere controllato in modo che tali errori possano essere rilevati dal compilatore.

La questione se l'eccezione sia "solitamente" evitata ha spesso a che fare con il costo e la convenienza di evitarla.

Ad esempio, è comodo e poco costoso determinare la dimensione di un vettore, chiamando il metodo `size`, che ritorna in tempo costante; pertanto, è probabile che il codice in uso utilizzi questo metodo per evitare la `IndexOutOfBoundsException`.

A volte, però, non esiste un modo conveniente per evitare l'eccezione e accade che evitare l'eccezione diventi costoso.

Entrambe le situazioni si presentano per la `search`. Potrebbe non esistere un'altra procedura per determinare se l'elemento si trova nell'array, poiché questo è (in parte) lo scopo di `search`. Inoltre, se tale procedura esistesse, la sua chiamata sarebbe costosa.

Quindi, per ricapitolare, si dovrebbe usare un'eccezione non controllata solo se si prevede che gli utenti scriveranno solitamente del codice che garantisca che l'eccezione non si verifichi, poiché:

- Esiste un modo comodo e poco costoso per evitare l'eccezione
- Il contesto di utilizzo è locale

Altrimenti, si dovrebbe usare un'eccezione controllata.

4.5 - Programmazione difensiva

Le eccezioni possono essere utilizzate per supportare la pratica della **programmazione difensiva**, ovvero scrivere ogni procedura per difendersi dagli errori. Gli errori possono essere introdotti da altre procedure, dall'hardware oppure dai dati immessi dall'utente; questi ultimi errori continueranno a esistere anche se il software è privo di errori. Un meccanismo di eccezione fornisce un mezzo per

trasmettere informazioni sugli errori e un modo per gestire gli errori senza ingombrare il flusso principale di una routine. Pertanto, incoraggia una metodologia di scrittura del codice che controlla i problemi e li segnala in modo ordinato.

Ad esempio, l'implementazione di una procedura con una clausola *requires* dovrebbe verificare, se possibile, se la clausola *requires* è soddisfatta. Ciò solleva la questione di cosa fare se la clausola *requires* non è soddisfatta. Una possibilità è quella di interrompere il programma con un messaggio di errore se il controllo fallisce. Tuttavia, questo non è un approccio molto robusto. È meglio utilizzare il meccanismo delle eccezioni perché, se la chiamata avviene in un contesto in cui un livello superiore può recuperare i problemi in modo generico (ad esempio, riavviando), sarà in grado di farlo anche per il controllo fallito.

È una buona idea avere un particolare tipo di eccezione dedicato a situazioni come il **mancato soddisfacimento della clausola *requires***. Un buon nome per questo tipo è `FailureException`; si tratta di un'eccezione non controllata. Le intestazioni delle procedure non devono elencare `FailureException`, così come le loro specifiche non dovrebbero menzionare la possibilità di lanciarla. Il motivo è che questa eccezione viene usata per situazioni che non corrispondono a quanto descritto nelle specifiche di una procedura. L'eccezione indica invece che qualcosa è rotto e che la procedura non è in grado di soddisfare le sue specifiche.

Altre situazioni in cui dovrebbe essere lanciata una `FailureException` è, ad esempio, l'utilizzo di `search` in un contesto in cui si sa che `x` è nell'array, ma la chiamata a `search` lancia `NotFoundException`. Poiché si tratta di un'eccezione verificata, è necessario catturarla; il codice può così lanciare `FailureException`. La stringa contenuta nella `FailureException`, come al solito, deve indicare il problema. Un modo semplice per farlo è concatenare le informazioni sulla classe e sul metodo con la stringa ottenuta da `NotFoundException`, ad esempio:

```
java
catch (NotFoundException e) {
    throw new FailureException("C.p" + e.toString());
}
```

Più in generale, `FailureException` dovrebbe essere sollevata ogni volta che il codice verifica un'assunzione che dovrebbe essere valida e scopre che non lo è.

Naturalmente, la verifica dei problemi richiede tempo e si è tentati di non preoccuparsi dei controlli, oppure di usarli solo durante il debug e di disabilitarli durante la produzione. In genere si tratta di una pratica poco saggia. La programmazione difensiva è particolarmente preziosa durante la produzione, perché può evitare che un piccolo errore causi un grosso problema, come un database danneggiato. I controlli dovrebbero essere disabilitati solo se si è dimostrato che gli errori non possono mai

verificarsi o se i controlli sono costosi.

5 - Astrazione di dati

L'astrazione dei dati ci permette di astrarre dai dettagli di come gli oggetti di dati sono implementati e di come gli oggetti si comportano. Questa attenzione al comportamento degli oggetti costituisce la base della programmazione orientata agli oggetti.

L'astrazione dei dati ci permette di estendere il linguaggio di programmazione in uso con nuovi **tipi di dati**. I nuovi tipi necessari dipendono dal **dominio applicativo** del programma (ad esempio, pile e tabelle di simboli sono utili nell'implementazione di un compilatore o di un interprete).

In ogni caso, l'astrazione dei dati consiste in un **insieme di oggetti**, ad esempio pile o polinomi, e in un **insieme di operazioni**.

Ad esempio, le operazioni sulle matrici includono l'addizione, la moltiplicazione e così via, mentre il deposito e il prelievo sono operazioni sui conti.

I nuovi tipi di dati dovrebbero incorporare l'astrazione sia per parametrizzazione che per specificazione.

- L'astrazione per parametrizzazione può essere ottenuta nello stesso modo delle procedure, utilizzando i parametri ovunque sia ragionevole farlo.
- L'astrazione per specificazione si ottiene rendendo le **operazioni parte del tipo**.

Per capire perché le operazioni sono necessarie, si consideri cosa succede se si considera un tipo come un semplice insieme di oggetti. In questo caso, tutto ciò che serve per implementare il tipo è scegliere una rappresentazione di memoria per gli oggetti; tutti i programmi in uso possono essere implementati con questa rappresentazione. Tuttavia, se la rappresentazione cambia, o anche se cambia la sua interpretazione, tutti i programmi che utilizzano il tipo devono essere modificati: non c'è modo di limitare l'impatto della modifica.

D'altra parte, supponiamo di includere le operazioni nel tipo, ottenendo **astrazione dei dati** = (**oggetti**, **operazioni**) e chiediamo agli utenti di chiamare le operazioni invece di accedere direttamente alla rappresentazione. In questo modo, per implementare il tipo, implementiamo le operazioni in termini della rappresentazione scelta e dobbiamo reimplementare le operazioni se cambiamo la rappresentazione. Tuttavia, non è necessario reimplementare i programmi che li utilizzano, perché non usano la rappresentazione.

Se vengono fornite abbastanza operazioni, la mancanza di accesso alla rappresentazione non causerà alcuna difficoltà agli utenti: tutto ciò che devono fare agli oggetti può essere fatto, e in modo efficiente, tramite chiamate alle operazioni. In generale, ci saranno operazioni per creare e modificare gli oggetti e per ottenere informazioni sui loro valori. Naturalmente, gli utenti possono aumentare l'insieme delle operazioni definendo procedure autonome, ma tali procedure non avrebbero accesso

alla rappresentazione.

L'astrazione dei dati ci permette di **rimandare le decisioni sulle strutture dei dati** fino a quando non se ne comprende appieno l'utilizzo. La scelta delle strutture dati giuste è fondamentale per ottenere un programma efficiente. In assenza di astrazione dei dati, le strutture di dati devono essere definite troppo presto; la struttura scelta potrebbe mancare delle informazioni necessarie o essere organizzata in modo inefficiente.


L'astrazione dei dati è preziosa anche durante la **modifica** e la **manutenzione del programma**. In questa fase è particolarmente probabile che le strutture dei dati cambino, sia per migliorare le prestazioni sia per adattarsi a requisiti mutevoli. L'astrazione dei dati limita le modifiche alla sola implementazione del tipo; non è necessario modificare nessuno dei moduli che lo utilizzano.

5.1 - Specifiche per le astrazioni di dati

Proprio come nel caso delle procedure, il significato di un tipo non dovrebbe essere dato da nessuna delle sue implementazioni. Invece, una specifica dovrebbe definire il suo **comportamento**. Poiché gli oggetti del tipo vengono utilizzati solo chiamando le operazioni, la maggior parte delle specifiche consiste nello spiegare cosa fanno le operazioni. In Java, i nuovi tipi sono definiti da classi o interfacce.

Ogni classe definisce un tipo definendo un **nome** per il tipo, un **insieme di costruttori** e un insieme di *istanze di metodi* o **metodi**.

In particolare:

- L'intestazione `class dname` indica che si sta definendo un nuovo tipo di dati chiamato `dname`. L'intestazione contiene una dichiarazione di visibilità della classe.
- La specifica si compone di tre parti.
 - La **panoramica** (**OVERVIEW**) fornisce una breve descrizione dell'astrazione dei dati, compreso un modo di vedere gli oggetti astratti in termini di concetti "ben compresi". Di solito presenta un modello per gli oggetti, cioè descrive gli oggetti in termini di altri oggetti che il lettore della specifica può aspettarsi di capire.
 Ad esempio, le pile possono essere definite in termini di sequenze matematiche. La sezione di panoramica indica anche se gli oggetti del tipo sono mutabili, in modo che il loro stato possa cambiare nel tempo, o immutabili.
 - La parte `constructors` della specifica definisce i **costruttori**. I costruttori vengono utilizzati per **inizializzare** nuovi oggetti del tipo, chiamate **istanze**. Una volta che un oggetto è stato creato (e inizializzato da un costruttore), gli utenti possono accedervi

chiamando i suoi metodi.

- La parte `methods` definisce i metodi che permettono di accedere agli oggetti una volta creati.

Ad esempio:

```
java
visibility class dname {
    // OVERVIEW: Una breve descrizione del comportamento degli oggetti del
    // tipo va qui

    // constructors
    // le specifiche dei costruttori vanno qua

    // methods
    // le specifiche dei metodi vanno qua
}
```

I costruttori e i metodi sono procedure e vengono specificati utilizzando la notazione delle specifiche, con la differenza che:

- I metodi e i costruttori appartengono entrambi agli oggetti, piuttosto che alle classi. Pertanto, la parola chiave `static` non apparirà nelle intestazioni dei metodi:
- L'oggetto a cui appartiene un metodo o un costruttore è disponibile come argomento implicito e questo oggetto può essere indicato nelle specifiche del metodo o del costruttore come `this`.

5.1.1 - Specifiche di `IntSet`

Gli `IntSet` sono insiemi non limitati di numeri interi con operazioni per creare un nuovo `IntSet` vuoto, verificare se un dato numero intero è un elemento di un `IntSet` e aggiungere o rimuovere elementi.

La specifica per l'astrazione dei dati `IntSet`:

```
public class IntSet {
    // OVERVIEW: Gli IntSet sono insiemi mutabili e non limitati di numeri
    // interi. Un tipico IntSet è {x1, . . . , xn}.

    // constructors
    public IntSet()
        // EFFECTS: Inizializza this per essere vuoto

    // methods
    public void insert(int x)
```

```

// MODIFIES: this
// EFFECTS: Aggiunge x agli elementi di this,
// i.e., this_post = this + { x }.

public void remove(int x)
    // MODIFIES: this
    // EFFECTS: Rimuove x da this, i.e., this_post = this - { x }.

public boolean isIn(int x)
    // EFFECTS: Se x è in this ritorna true, altrimenti ritorna false

public int size()
    // EFFECTS: Ritorna la cardinalità di this

public int choose() throws EmptyException
    // EFFECTS: Se this è vuoto, lancia EmptyException, altrimenti
    // ritorna un elemento arbitrario di this.

}

```

La *panoramica* indica che gli `IntSet` sono mutabili. Indica anche che modelliamo `IntSet` in termini di insieme matematici. Nel resto della specifica, specificheremo ogni operazione utilizzando questo modello. Viene utilizzata la notazione degli insiemi nelle specifiche dei metodi.

In particolare:

- Un insieme è denotato come $\{x_1, \dots, x_n\}$. L' x_i -esimo è un elemento dell'insieme. Non esistono duplicati in un insieme.
- **Unione di insiemi:** $t = s_1 + s_2$ è l'insieme che contiene tutti gli elementi dell'insieme s_1 e tutti gli elementi dell'insieme s_2 . Se s_1 e s_2 contengono un elemento in comune, ci sarà solo un'occorrenza di tale elemento in t
- **Differenza tra gli insiemi:** $t = s_1 - s_2$ è un'insieme che contiene tutti gli elementi di s_1 che non sono elementi di s_2
- **Intersezione di insiemi:** $t = s_1 \& s_2$ è un'insieme che contiene tutti gli elementi che sono sia in s_1 che s_2
- **Cardinalità:** $|s|$ indica la dimensione dell'insieme s
- **Appartenenza a un insieme:** $x \text{ in } s$ è vero se x è un elemento di s
- **Insieme "former":** $t = \{x \mid p(x)\}$ è l'insieme di tutti gli elementi x per il quale $p(x)$ è vero

Il tipo `IntSet` ha un singolo **costruttore** che inizializza il nuovo insieme come vuoto; si noti che la specifica si riferisce al nuovo insieme come `this`. Poiché un costruttore modifica sempre `this` (per

inizializzalo), non ci si preoccupa di indicare la modifica nella clausola *modifies*. Infatti, questa modifica è invisibile per gli utenti: essi non hanno accesso all'oggetto del costruttore se non dopo l'esecuzione del costruttore stesso e quindi non possono osservare il cambiamento di stato.

Una volta che un oggetto `IntSet` esiste, gli elementi possono essere aggiunti chiamando il suo metodo `insert` e gli elementi possono essere rimossi chiamando `remove`; ancora una volta, le specifiche si riferiscono all'oggetto come `this`. Questi metodi sono **mutatori**, poiché modificano lo stato del loro oggetto; le loro specifiche chiariscono che si tratta di mutatori, poiché contengono una clausola *modifies* che indica che `this` viene modificato. Si noti che le specifiche di `insert` e `remove` utilizzano la notazione `this_post` per indicare il valore di `this` al ritorno dell'operazione. Un nome di argomento di input senza il qualificatore *post* indica sempre il valore quando l'operazione viene chiamata.

I metodi rimanenti sono **osservatori**: restituiscono informazioni sullo stato del loro oggetto, ma non lo cambiano. Gli osservatori non hanno una clausola *modifies* che dichiara che `this`, o qualche oggetto del suo tipo, viene modificato; tuttavia, gli osservatori in genere non modificano nulla.

Il metodo `choose` restituisce un elemento arbitrario dell'`IntSet`, quindi è sottodeterminato. Lancia un'eccezione se l'insieme è vuoto. L'eccezione può essere eliminata, in quanto l'utente può chiamare il metodo `size` prima di chiamare `choose` per assicurarsi che l'insieme non sia vuoto in modo economico e conveniente.

Nota bene: Si noti che `insert` non lancia un'eccezione se l'intero è già presente nell'insieme e, analogamente, `remove` non lancia un'eccezione se l'intero non è presente nell'insieme. Queste decisioni si basano su ipotesi di utilizzo degli insiemi. Ci aspettiamo che gli utenti aggiungano e rimuovano elementi dell'insieme senza preoccuparsi che siano già presenti. Pertanto, i metodi non lanciano eccezioni. Se ci aspettassimo un modello di utilizzo diverso, potremmo cambiare le specifiche e le intestazioni di questi metodi (per lanciare un'eccezione), oppure potremmo fornire altri metodi che lanciano un'eccezione (ad esempio, `insertNonDup` e `removeIfIn`), in modo che gli utenti possano scegliere il metodo che meglio si adatta alle loro esigenze.

Nella specifica `IntSet` si fa affidamento sul fatto che il lettore sappia cosa sono gli insiemi matematici, altrimenti la specifica non sarebbe comprensibile. In generale, questo affidamento alla descrizione informale è un punto debole delle specifiche informali.

Nota bene: la specifica assume la forma di una versione preliminare della classe.

5.1.2 - L'astrazione Poly

I `Poly` sono polinomi con coefficienti interi. A differenza degli `IntSet`, i `Poly` sono immutabili: una volta che un `Poly` è stato creato (e inizializzato da un costruttore), non può essere modificato. Sono previste operazioni per creare un `Poly` a un termine e per aggiungere, sottrarre e moltiplicare i `Poly`.

```
public class Poly {
    // OVERVIEW: Polys sono polinomi immutabili con coefficienti interi.
    // Un tipico Poly è c0+c1x+...

    // constructors
    public Poly()
        // EFFECTS: Inizializza this per essere un polinomio zero.

    public Poly(int c, int n) throws NegativeExponentException
        // EFFECTS: Se n < 0 lancia NegativeExponentException, altrimenti
        // inizializza this per essere il Poly cx^n

    // methods
    public int degree()
        // EFFECTS: Ritorna il grado di this, i.e., il più grande esponente
        // con un coefficiente diverso da zero. Ritorna 0 se this è un
        // Poly zero.

    public int coeff(int d)
        // EFFECTS: Ritorna il coefficiente del termine di this il cui
        // esponente è d

    public Poly add(Poly q) throws NullPointerException
        // EFFECTS: Se q è nullo lancia NullPointerException, altrimenti
        // ritorna il Poly this + q

    public Poly mul(Poly q) throws NullPointerException
        // EFFECTS: Se q è nullo lancia NullPointerException, altrimenti
        // ritorna il Poly this * q

    public Poly sub(Poly q) throws NullPointerException
        // EFFECTS: Se q è nullo lancia NullPointerException, altrimenti
        // ritorna il Poly this - q

    public Poly minus ( )
        // EFFECTS: Ritorna il Poly - this
}
```

```
}
```

Il tipo `Poly` ha due **costruttori**, uno per creare il polinomio zero e uno per creare un monomio arbitrario. In generale, un tipo può avere un certo numero di costruttori. Tutti i costruttori hanno lo stesso nome, il nome del tipo, e quindi, se c'è più di un costruttore, questo nome è *sovraccarico*; le due definizioni per il costruttore `Poly` sono legali, poiché una non ha argomenti e l'altra ne ha due. `Poly` non ha metodi mutatori: nessun metodo ha una clausola di *modifica*. Questo è ciò che ci aspettiamo di vedere per un'astrazione di dati immutabili. Inoltre, le specifiche dei metodi non utilizzano la notazione *post* che è stata utilizzata nella specifica `IntSet`. Questa notazione non è necessaria per le astrazioni immutabili: poiché lo stato degli oggetti non cambia, gli stati pre e post degli oggetti sono identici.

Nella definizione di `Poly`, dobbiamo decidere se l'eccezione `NegativeExponentException` sia controllata o meno. Poiché sembra probabile che gli utenti evitino le chiamate con esponente negativo, è opportuno che l'eccezione sia non controllata.

5.2 - Utilizzo delle astrazioni di dati

Alcuni esempi di procedure che utilizzano astrazioni di dati:

```
java
public static Poly diff(Poly p) throws NullPointerException {
    // EFFECTS: Se p è nullo lancia NullPointerException, altrimenti
    // ritorna Poly ottenuto dalla differenza con p
    Poly q = new Poly();
    for (int i = 1; i <= p.degree(); i++)
        q = q.add(new Poly(p.coeff(i)*i, i-1));
    return q;
}

public static IntSet getElements(int[] a) throws NullPointerException {
    // EFFECTS: Se a è nullo lancia NullPointerException, altrimenti
    // ritorna un insieme contenente un'entrata per ogni elemento distinto
    // di a
    IntSet s = new IntSet();
    for (int i = 0; i < a.length; i++) s.insert(a[i]);
    return s;
}
```

- Il metodo `diff` restituisce un nuovo `Poly` che è il risultato della differenziazione del suo argomento `Poly`.
- La routine `getElements` restituisce un `IntSet` contenente gli interi del suo argomento

array `a` ; non ci sono duplicati nell'insieme restituito (poiché gli insiemi non contengono duplicati) anche se ci sono duplicati tra gli elementi di `a` .

Queste routine sono scritte in base alle specifiche delle astrazioni utilizzate e possono utilizzare solo ciò che è descritto nelle specifiche. Non sono in grado di accedere ai dettagli di implementazione degli oggetti astratti poiché, come vedremo, questo accesso è limitato alle implementazioni dei costruttori e dei metodi degli oggetti. Possono usare i metodi per accedere allo stato dell'oggetto e per modificarlo, se l'oggetto è mutabile, e possono usare i costruttori per inizializzare nuovi oggetti.

5.3 - Implementazione delle astrazioni di dati

Una classe definisce un nuovo tipo e ne fornisce un'implementazione. La specifica costituisce la definizione del tipo. Il resto della classe fornisce l'implementazione.

Per implementare un'astrazione di dati, si sceglie una **rappresentazione**, o **rep**, per i suoi oggetti e si implementano i costruttori per inizializzare correttamente la rappresentazione e i metodi per utilizzare/modificare correttamente la rappresentazione. La rappresentazione scelta deve permettere di implementare tutte le operazioni in modo estremamente semplice ed efficiente.

Ad esempio, una rappresentazione plausibile per un oggetto `IntSet` è un vettore, dove ogni intero dell' `IntSet` è un elemento del vettore. Potremmo scegliere se fare in modo che ogni elemento dell'insieme si presenti esattamente una volta nel vettore o se permettere che si presenti più volte. Quest'ultima scelta rende più veloce l'implementazione di `insert` , ma rallenta `remove` e `isIn` . Poiché è probabile che `isIn` venga richiamato di frequente, sceglieremo la prima opzione e quindi non ci saranno elementi duplicati nel vettore.

5.3.1 - Implementazione delle astrazioni di dati in Java

Una rappresentazione ha tipicamente un certo numero di componenti; in Java, ognuno di questi è una **variabile di istanza della classe** che implementa l'astrazione dei dati. Le implementazioni dei costruttori e dei metodi accedono e manipolano le variabili di istanza.

Pertanto, se considerati dal punto di vista dell'implementazione, gli oggetti hanno sia metodi che variabili di istanza. Per supportare l'astrazione, tuttavia, è importante limitare l'accesso alle variabili di istanza all'implementazione dei metodi e dei costruttori; ciò consente, ad esempio, di reimplementare un tipo astratto senza influenzare il codice che utilizza il tipo. Pertanto, le **variabili di istanza non devono essere visibili agli utenti**; il codice che utilizza gli oggetti può fare riferimento solo ai loro metodi. Per evitare che le variabili di istanza siano visibili agli utenti, è necessario dichiararle **private**. Inoltre, Java consente alle variabili di istanza di avere una visibilità diversa da quella privata. In generale, non è una buona idea avere variabili di istanza pubbliche.

Tuttavia, è possibile dichiarare variabili statiche all'interno di una classe. Tali variabili appartengono alla classe stessa, piuttosto che a oggetti specifici, proprio come i metodi statici appartengono alla

classe. Però, le variabili statiche non vengono utilizzate molto spesso nell'implementazione di astrazioni di dati.

5.3.2 - Implementazione di IntSet

```
public class IntSet {
    // OVERVIEW: Gli IntSet sono insiemi non limitati e mutabili di numeri
    // interi.
    private Vector els; // the rep

    // constructors
    public IntSet() {
        // EFFECTS: Inizializza this per essere vuoto
        els = new Vector();
    }

    // methods
    public void insert(int x) {
        // MODIFIES: this
        // EFFECTS: Aggiungi x agli elementi di this
        Integer y = new Integer(x);
        if (getIndex(y) < 0) els.add(y);
    }

    public void remove(int x) {
        // MODIFIES: this
        // EFFECTS: Rimuovi 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: Ritorna true se x è in this, altrimenti ritorna false
        return getIndex(new Integer(x)) >= 0;
    }

    private int getIndex(Integer x) {
        // EFFECTS: Se x è in this ritorna l'indice dove x appare altrimenti
        // ritorna -1
        for (int i = 0; i < els.size(); i++)
            if (x.equals(els.get(i))) return i;
    }
}
```

```

        return -1;
    }

    public int size() {
        // EFFECTS: Ritorna la cardinalità di this
        return els.size();
    }

    public int choose() throws EmptyException {
        // EFFECTS: Se this è vuoto lancia EmptyException, altrimenti
        // ritorna un elemento arbitrario di this
        if (els.size() == 0) throw new EmptyException("IntSet.choose");
        return els.lastElement();
    }
}

```

- Da notare, la definizione della **rep** di `IntSet`, che precede le implementazioni dei costruttori e dei metodi. In questo caso, la **rep** consiste in una singola variabile di istanza. Poiché questa variabile ha visibilità privata, è accessibile solo al codice interno alla sua classe.
- I costruttori e i metodi appartengono a un particolare oggetto del loro tipo. L'oggetto viene passato come argomento aggiuntivo e implicito ai costruttori e ai metodi, che possono farvi riferimento utilizzando la parola chiave `this`.

Ad esempio, si può accedere alla variabile di istanza `els` utilizzando la forma `this.els`. Tuttavia, il prefisso non è necessario: il codice può fare riferimento ai metodi e alle variabili di istanza del proprio oggetto utilizzando semplicemente i loro nomi. Pertanto, nei metodi e nei costruttori della figura, `els` si riferisce alla variabile di istanza `els` di `this`.

L'implementazione di `IntSet` è semplice. Il costruttore inizializza l'oggetto creando il vettore che conterrà gli elementi e assegnandolo a `els`; poiché il vettore è vuoto, non è necessario fare altro. I metodi `insert`, `remove` e `isIn` utilizzano tutti il metodo privato `getIndex` per determinare se l'elemento di interesse è già presente nell'insieme. Questo controllo consente a `insert` di preservare la condizione di non duplicazione. Questa condizione viene rispettata in `size` (perché altrimenti la dimensione del vettore non sarebbe uguale a quella dell'insieme) e in `remove` (perché altrimenti potrebbero esserci altre occorrenze dell'elemento che andrebbero rimosse).

Si noti che `getIndex` ha visibilità privata; pertanto, non può essere chiamato al di fuori della classe. Il progetto sfrutta questo fatto facendo in modo che `getIndex` restituisca `-1` quando l'elemento non è presente nel vettore, anziché utilizzare un'eccezione. Come discusso nel Capitolo 4, si tratta di un approccio soddisfacente, poiché `getIndex` viene utilizzato solo all'interno di questa classe.

Poiché i vettori non possono memorizzare gli `int`, i metodi utilizzano gli oggetti `Integer` per contenere gli elementi dell'insieme. Questo approccio è piuttosto scomodo. Un'alternativa è l'uso di

matrici di `int`; ma questo comporta delle difficoltà, poiché l'implementazione di `IntSet` dovrebbe passare a matrici più grandi man mano che l'insieme cresce. L'implementazione di `Vector` si occupa di questo problema in modo efficiente.

Nota bene: `getIndex` utilizza un metodo `equals` per verificare l'appartenenza. Questo controllo è corretto perché `equals` per gli oggetti `Integer` restituisce `true` solo se i due oggetti da confrontare sono entrambi `Integer` ed entrambi contengono lo stesso valore intero.

5.3.3 - Implementazione di Poly

```
public class Poly {
    // OVERVIEW: ...
    private int[] trms;
    private int deg;

    // constructors
    public Poly() {
        // EFFECTS: Inizializza this per essere un polinomio zero
        trms = new int[1]; deg = 0;
    }

    public Poly(int c, int n) throws NegativeExponentException {
        // EFFECTS: Se n < 0 lancia NegativeExponentException, altrimenti
        // inizializza this per essere il Poly cx^n
        if (n < 0)
            throw new NegativeExponentException(
                "Poly(int, int) constructor");
        if (c == 0) { trms = new int[1]; deg = 0; return; }
        trms = new int[n+1];
        for (int i = 0; i < n; i++) trms[i] = 0;
        trms[n] = c;
        deg = n;
    }

    private Poly(int n) { trms = new int[n+1]; deg = n;}

    // methods
    public int degree() {
        // EFFECTS: Ritorna il grado di this, i.e., il più grande esponente
        // con un coefficiente diverso da zero. Ritorna 0 se this è un
        // Poly zero
        return deg;
    }
}
```

```

}

public int coeff(int d) {
    // EFFECTS: Ritorna il coefficiente del termine di this il cui
    // esponente è d
    if (d < 0 || d > deg) return 0; else return trms[d];
}

public Poly sub(Poly q) throws NullPointerException {
    // EFFECTS: Se q è nullo lancia NullPointerException, altrimenti
    // ritorna il Poly this - q
    return add(q.minus());
}

public Poly minus( ) {
    // EFFECTS: Ritorna il Poly -this
    Poly r = new Poly(deg);
    for (int i = 0; i < deg; i++) r.trms[i] = - trms[i];
    return r;
}

public Poly add(Poly q) throws NullPointerException {
    // EFFECTS: Se q è nullo lancia NullPointerException, altrimenti
    // ritorna il Poly this + q
    Poly la, sm;
    if (deg > q.deg) {la = this; sm = q;} else {la = q; sm = this;}
    int newdeg = la.deg // nuovo grado è il grado più grande
    if (deg == q.deg) // a meno che non ci siano zeri finali
        for (int k = deg; k > 0; k--)
            if (trms[k] + q.trms[k] != 0) break; else newdeg--;
    Poly r = new Poly(newdeg); // prendere un nuovo Poly
    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;
}

public Poly mul(Poly q) throws NullPointerException {
    // EFFECTS: Se q è null lancia NullPointerException, altrimenti
    // ritorna il Poly this*q
    if ((q.deg == 0 && q.trms[0] == 0) || (deg == 0 && trms[0] == 0))
        return new Poly();
}

```



```

    Poly r = new Poly(deg+q.deg);
    r.trms[deg+q.deg] = 0; // preparazione per il calcolo i coefficienti
    for (int i = 0; i <= deg; j++)
        r.trms[i+j] = r.trms[i+j] + trms[i]*q.trms[j];
    return r;
}
}

```

A differenza degli `IntSet`, i `Poly` sono immutabili e quindi la loro dimensione non cambia nel corso del tempo. Pertanto, possiamo rappresentare una `Poly` come una matrice piuttosto che come un vettore. L'elemento *i*-esimo dell'array conterrà il coefficiente dell'esponente *i*-esimo; questa rappresentazione ha senso solo se il `Poly` è denso. Il `Poly` zero viene rappresentato come una matrice a un elemento contenente zero. Inoltre, avremo una variabile di istanza che tiene traccia del grado della `Poly`, poiché è conveniente.

Nota bene:

- Molti metodi, come `add` e `mul` utilizzano variabili di istanza di altri oggetti `Poly` oltre al proprio oggetto. Il codice in un metodo può accedere a informazioni private di altri oggetti della sua classe e a informazioni private del suo stesso oggetto. Si noti come `sub` e `mul` siano implementati in termini di altri metodi `Poly`.
- Un altro punto è l'uso del costruttore `Poly` nelle implementazioni di `add`, `mul` e `minus`. Tutti questi metodi inizializzano in realtà la nuova `Poly`; ciò è consentito in quanto la nuova `Poly` è solo un altro oggetto della classe, a cui si può accedere nel metodo. Questi metodi creano il nuovo `Poly` utilizzando il costruttore privato (che non può essere chiamato dagli utenti) per ottenere un array della giusta dimensione. Nel caso di `mul`, ci si affida al fatto che il costruttore di array inizializza tutti gli elementi di un array di `int` a zero. Si noti anche la cura con cui ci si assicura che il nuovo oggetto `Poly` sia della dimensione giusta. Ciò richiede un pre-compito nel metodo `add` per gestire il caso degli zeri finali.

5.3.4 - Records

Supponiamo che i polinomi siano radi piuttosto che densi. In questo caso, l'implementazione precedente non sarebbe una buona soluzione, poiché è probabile che l'array sia grande e pieno di zeri. Invece, vorremmo memorizzare le informazioni solo per i coefficienti non nulli.

A tal fine si possono utilizzare due vettori:

```

private Vector coeffs; // i coefficienti non-zero
private Vector exps; // gli esponenti associati

```

Tuttavia, l'implementazione in questo caso deve garantire che i due array siano allineati. Sarebbe più comodo se si potesse utilizzare un solo vettore, ognuno dei cui elementi contenga sia il coefficiente che l'esponente associato. Questo può essere ottenuto utilizzando un **record**.

Un **record** è semplicemente un **insieme di campi**, ciascuno con un nome e un tipo. La classe che implementa questo tipo ha una variabile di istanza pubblica o visibile *dal* pacchetto per ogni campo; la visibilità del pacchetto significa che i campi possono essere accessibili da altro codice dello stesso pacchetto, ma non da altri.

La classe fornisce un costruttore per creare un nuovo oggetto del tipo; il costruttore prende argomenti per definire i valori iniziali dei campi.

Un esempio:

```
java
class Pair {
    // OVERVIEW: Un tipo record
    int coeff;
    int exp;

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

Poiché non è stata indicata esplicitamente alcuna visibilità per la classe e le sue variabili di istanza, esse sono visibili come pacchetto.

Possiamo utilizzare `Pair` in un'implementazione di polinomi sparsi:

```
private Vector trms; // i termini con coefficiente no-zero
```

Qui ogni elemento di `trms` è un `Pair`. Questa rappresentazione è più semplice di quella che utilizza due vettori. Un ulteriore vantaggio è che ci permette di evitare l'uso del metodo `intValue`.

Ad esempio, se utilizziamo due vettori abbiamo:

```
java
public int coeff(int x) {
    for (int i = 0; i < exps.size(); i++)
        if (((Integer) exps.get(i)).intValue() == x)
            return ((Integer) coeffs.get(i)).intValue();
    return 0;
}
```

Se utilizziamo il vettore di coppie, abbiamo:

```
java
public int coeff (int x) {
    for (int i = 0; i < trms.size(); i++) {
        Pair p = (Pair) trms.get(i);
```

```

        if (p.exp == x) return p.coef;
    }
    return 0;
}

```

5.4 - Metodi aggiuntivi

Tutti gli oggetti hanno metodi aggiuntivi, definiti da `Object`. Tutte le classi definiscono sottotipi di `Object` e quindi devono fornire tutti i metodi di `Object`. Inoltre, le classi erediteranno le implementazioni di questi metodi, a meno che non li implementino esplicitamente. L'ereditarietà dei metodi degli oggetti va bene se l'implementazione ereditata è corretta per il nuovo tipo; altrimenti, la classe deve fornire la propria implementazione.

Analizziamo i metodi `equals`, `clone` e `toString`.

5.4.1 - Metodi aggiuntivi: `equals`, `hashCode` e `similar`

Due oggetti dovrebbero essere `equals` se sono equivalenti dal punto di vista comportamentale. Ciò significa che non è possibile distinguere tra di loro utilizzando una qualsiasi sequenza di chiamate ai metodi degli oggetti.

- Nel caso di oggetti mutabili, tutti gli oggetti distinti sono distinguibili (cioè, `equals` ha lo stesso significato di `==`). In altre parole, gli oggetti mutabili sono `uguali` solo se sono lo stesso oggetto.

Ad esempio:

```

java
IntSet s = new IntSet();
IntSet t = new IntSet();
if (s.equals(t)) ...; else ...

```

Al momento dell'esecuzione dell'`if`, sia `s` che `t` hanno lo stesso stato (l'insieme vuoto).

Tuttavia, `s` e `t` sono comunque distinguibili, a causa delle mutazioni;

- Gli oggetti immutabili sono uguali se hanno lo stesso stato, perché non ci sarà modo di distinguerli chiamando i loro metodi.

Ad esempio:

```

java
Poly p = new Poly(3, 4);
Poly q = new(3, 4);
if (p.equals(q)) ...; else ...

```

Quando l'istruzione `if` viene eseguita, `p` e `q` hanno lo stesso stato (il polinomio $3x^4$).

Inoltre, poiché i polinomi sono immutabili, `p` e `q` avranno sempre lo stesso stato. Pertanto, la chiamata `p.equals(q)` nell'istruzione `if` dovrebbe restituire `true`.

Nota bene: L'implementazione predefinita di `equals` fornita da `Object` verifica se i due oggetti hanno la stessa identità. Questo è il test giusto per `IntSet`, ma è un test sbagliato per `Poly` e lo sarà per qualsiasi tipo immutabile. Quindi, quando si definisce un tipo immutabile, occorre fornire la propria implementazione di `equals`. Tuttavia, non ci si deve preoccupare di `equals` per i tipi mutabili; gli oggetti di questi tipi avranno un metodo `equals`, cioè quello ereditato da `Object`.

Ad esempio, nel caso da classe immutabile `Poly`, l'implementazione sarà:

```
java
public class Poly {
    // tutto quello che c'è scritto prima, più

    public boolean equals(Poly q) {
        if (q == null || deg != q.deg) return false;
        for (int i = 0; i <= deg; i++)
            if (trms[i] != q.trms[i]) return false;
        return true;
    }

    public boolean equals(Object z) {
        if (!(z instanceof Poly)) return false;
        return equals((Poly) z);
    }
}
```

`Poly` fornisce due definizioni per `equals`, uno sovrascrivendo il metodo di `Object` e uno extra:

```
java
boolean equals(Object) // intestazione del metodo di Object
boolean equals(Poly) // intestazione del metodo di Poly
```

La seconda è un'ottimizzazione; evita il cast e la chiamata su `instanceof`, che sono costosi, in contesti in cui sia l'oggetto che l'argomento sono noti al compilatore come `Poly`.

Nota bene: Si consideri

```
java
Poly x = new Poly(3, 7);
Object y = new Poly(3, 7);
.
```

```
.  
.   
if (x.equals(new Poly(3, 7)) ...  
if (x.equals(y)) ...
```

Nella prima istruzione `if`, la chiamata andrà all'implementazione ottimizzata di `equals` perché il compilatore sa che sia `x` che l'argomento sono `Poly`, ma la seconda chiamata andrà all'implementazione non ottimizzata perché il compilatore non sa che `y` è una `Poly`.

`Object` fornisce anche un metodo `hashCode`. Le specifiche di `hashCode` indicano che se due oggetti sono equivalenti secondo il metodo `equals`, `hashCode` dovrebbe produrre lo stesso valore per loro. Tuttavia, l'implementazione predefinita di `hashCode` non lo farà per i tipi immutabili. `hashCode` è necessario solo per i tipi destinati a essere chiavi di tabelle `hash`. Se il vostro tipo immutabile è uno di questi, dovete implementare `hashCode` in modo da rispettare questo vincolo sul suo comportamento.

Esiste una nozione di uguaglianza più debole che chiameremo *somiglianza*. Due oggetti sono simili se non è possibile distinguerli utilizzando alcun osservatore del loro tipo. Così come è utile avere un nome standard `equals` per il metodo che esegue i test di equivalenza, è altrettanto utile avere un nome standard per il metodo che esegue i test di somiglianza. Chiameremo questo metodo `similar`. Non è necessario fornire questo metodo in un nuovo tipo, ma è possibile farlo se lo si desidera. Per i tipi immutabili, simile e uguale sono la stessa cosa. Tuttavia, per i tipi mutabili, la somiglianza è più debole dell'equivalenza.

Ad esempio:

```
java  
IntSet s = new IntSet();  
IntSet t = new IntSet();  
if (s.similar(t)) ...; else ...
```

la chiamata `similar` dovrebbe restituire `true`.

5.4.2 - Metodi aggiuntivi: `clone`

Il metodo `clone` crea una copia del suo oggetto. La copia prodotta deve avere lo stesso stato del suo oggetto, cioè deve essere simile all'oggetto clonato. L'implementazione predefinita fornita da `Object` assegna semplicemente le variabili di istanza del vecchio oggetto a quelle del nuovo. Spesso questa non è un'implementazione corretta.

Ad esempio, nel caso di `IntSet`, le componenti `els` dei due oggetti condividerebbero lo stesso vettore. Quindi, quando viene effettuata una modifica a uno di essi (ad esempio, un

inserimento), anche lo stato dell'altro cambierà, il che non è corretto. D'altra parte, l'implementazione predefinita è corretta per `Poly`; anche in questo caso c'è una condivisione (dell'array che è il componente `trms`), ma la condivisione non ha importanza perché quell'array non viene mai modificato.

Se si vuole che un tipo fornisca un metodo `clone`, è necessario fornire la propria implementazione, se quella predefinita non è corretta. In generale, l'implementazione predefinita sarà corretta per i tipi immutabili e non corretta per quelli mutabili. Se l'implementazione predefinita è corretta, è possibile ereditarla inserendo `implements Cloneable` nell'intestazione della classe. Se una classe non include questa clausola nella sua intestazione né fornisce un'implementazione di `clone`, se il metodo `clone` viene chiamato su uno dei suoi oggetti, il codice lancerà `CloneNotSupportedException`.

Per esempio, le implementazioni di `IntSet` e `Poly` mostrate prima non supportano il clone e, pertanto, se il metodo `clone` viene chiamato su un oggetto `IntSet` o `Poly`, verrà sollevata la `CloneNotSupportedException`. Se volessimo che questi tipi fornissero il clone, dovremmo reimplementarlo per `IntSet`, ma potremmo eritarlo per `Poly`.

In particolare, per `Poly`, l'implementazione sarà:

```
java
public class Poly implements Cloneable {
    // ...
}
```

Mentre, per `IntSet` è:

```
java
public class IntSet() {
    // come scritto precedentemente, più

    private IntSet(Vector v) { els = v; }

    public Object clone() {
        return new IntSet((Vector) els.clone());
    }
}
```

Si noti che l'implementazione utilizza un costruttore aggiuntivo, in modo da poter inizializzare l'oggetto appena creato con il vettore giusto; poiché questo costruttore è privato, può essere chiamato solo all'interno della classe.

La *signature* del metodo `clone` in un sottotipo deve essere identica alla firma di `clone` per `Object`:

```
Object clone();
```

Purtroppo, questo significa che le chiamate al metodo non sono molto convenienti o efficienti.

Ad esempio, molto probabilmente chi chiama `s.clone()`, dove `s` è un `IntSet`, vuole ottenere come risultato un oggetto `IntSet`. In effetti, il metodo `clone` di `IntSet` produce un oggetto `IntSet`. Tuttavia, il tipo di ritorno di `clone` indica che viene restituito un `Object`. Poiché `Object` non è un sottotipo di `IntSet` (anzi, è vero il contrario), l'oggetto restituito da `clone` non può essere assegnato a una variabile `IntSet`; il chiamante deve invece eseguire il cast del risultato, ad esempio,

```
java
IntSet t = (IntSet) s.clone();
```

5.4.3 - Metodi aggiuntivi: `toString`

Il metodo `toString` produce una stringa che rappresenta lo stato attuale del suo oggetto, insieme a un'indicazione del suo tipo.

Ad esempio, per un `IntSet` e un `Poly`, si potrebbe voler vedere una rappresentazione del tipo:

- `IntSet: {1, 7, 3}`
- `Poly: 2 + 3x + 5x**2`

L'implementazione di `toString` fornita da `Object` non è molto informativa: fornisce il nome del tipo di oggetto e il suo codice hash. Pertanto, quasi ogni tipo dovrebbe fornire la propria implementazione di `toString`.

Ad esempio, per `IntSet`:

```
java
public String toString() {
    if (els.size() == 0) return "IntSet:{}";
    String s = "IntSet: {" + els.elementAt(0).toString();
    for (int i = 1; i < els.size(); i++)
        s = s + " , " + els.elementAt(i).toString();
    return s + "}";
}
```

5.5 - Ausili per la comprensione delle implementazioni

In questa sezione discutiamo due informazioni, la funzione di astrazione e l'invariante di rappresentazione, che sono particolarmente utili per comprendere l'implementazione di un'astrazione

di dati.

La **funzione di astrazione** cattura l'intento del progettista nella scelta di una particolare rappresentazione. È la prima cosa che si decide quando si inventa il rappresentante: quali variabili di istanza usare e come si relazionano con l'oggetto astratto che intendono rappresentare. La funzione di astrazione descrive semplicemente questa decisione.

L'**invariante di rappresentazione** viene creato mentre si studia come implementare i costruttori e i metodi. Cattura i presupposti comuni su cui si basano queste implementazioni; in questo modo, permette di considerare l'implementazione di ogni operazione in modo isolato dalle altre.

La funzione di astrazione e l'invariante di rappresentazione insieme forniscono una documentazione preziosa, sia per l'implementatore originale sia per chi legge il codice.

Data la loro utilità, sia la funzione di astrazione che l'invariante di rappresentazione dovrebbero essere inclusi come commenti nel codice.

5.5.1 - La funzione di astrazione

Qualsiasi implementazione di un'astrazione di dati deve definire come vengono rappresentati gli oggetti appartenenti al tipo. Nella scelta della rappresentazione, l'implementatore ha in mente una relazione tra i rappresentanti e gli oggetti astratti.

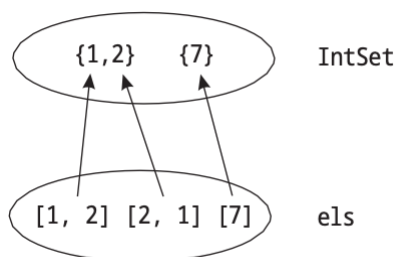
Ad esempio, nell'implementazione degli `IntSet`, questi sono rappresentati da vettore, dove gli elementi del vettore sono gli elementi dell'insieme.

Questa relazione può essere definita da una funzione chiamata **funzione di astrazione** che mappa dalle variabili di istanza che costituiscono la rappresentazione di un oggetto all'oggetto astratto rappresentato.

$AF: C \rightarrow A$

In particolare, la funzione di astrazione `AF` mappa uno **stato concreto**, cioè lo stato di un oggetto della classe `C`, a uno **stato astratto**, cioè lo stato di un oggetto astratto. Per ogni oggetto `c` appartenente a `C`, `AF(c)` è lo stato dell'oggetto astratto `a`, appartenente a `A`, che `c` rappresenta.

Ad esempio, la funzione di astrazione per l'implementazione di `IntSet` mappa le variabili di istanza degli oggetti della classe `IntSet` agli stati astratti di `IntSet`:



Questa funzione di astrazione è di tipo *multi-a-uno*: molti componenti `els` corrispondono allo stesso elemento astratto. Ad esempio, l'`IntSet {1, 2}` è rappresentato da un oggetto il cui vettore `els` contiene l'intero di valore 1 seguito dall'intero di valore 2, ma anche da un oggetto il

cui vettore `els` contiene i due interi nell'ordine opposto. Poiché il processo di astrazione comporta l'oblio di informazioni irrilevanti, non sorprende che le funzioni di astrazione siano spesso molte a una. In questo esempio, l'ordine in cui gli elementi appaiono nel componente `els` è irrilevante.

La funzione di astrazione è un'informazione cruciale su un'implementazione. Definisce il **significato della rappresentazione**, il modo in cui gli oggetti della classe devono implementare gli oggetti astratti.

Nella stesura di una descrizione di questo tipo, tuttavia, siamo ostacolati dal fatto che se la specifica del tipo è informale, l'intervallo della funzione di astrazione non è realmente definito.

Supereremo questo problema fornendo una descrizione di un oggetto astratto "tipico". Questo ci permette di definire la funzione di astrazione in termini di questo oggetto tipico. La descrizione dello stato dell'oggetto astratto tipico fa parte della specifica; viene fornita nella sezione panoramica.

Ad esempio, l'**OVERVIEW** di `IntSet` afferma che:

```
java
// Un IntSet tipico è {x1, ..., xn}
```

Ricordiamo che stiamo utilizzando gli insiemi matematici per indicare gli stati di `IntSet`:

```
java
// La funzione di astrazione è
// AF(c) = {c.els[i].intValue | 0 <= i < c.els.size}
```

La notazione $\{x|p(x)\}$ descrive l'insieme di tutti gli x tali che il predicato $p(x)$ è vero. L'`AF`, in questo caso, dice che gli elementi dell'insieme sono esattamente i valori interi contenuti nel vettore `els`. Nel definire la funzione di astrazione, utilizziamo alcune comode abbreviazioni: usiamo la notazione `c.els[i]` per indicare l'uso del metodo `get` di `Vector` e omettiamo la `()` quando usiamo metodi senza argomenti (come `intValue` e `size`). Si omette anche il casting e si assume semplicemente che gli elementi di il vettore `els` sono numeri interi.

Come secondo esempio, consideriamo l'implementazione di `Poly`. Abbiamo scelto di rappresentare un `Poly` come un array in cui l'elemento i -esimo contiene il coefficiente i -esimo fino al grado. Possiamo descrivere questa rappresentazione come segue:

```
java
// Un tipico Poly è c0 + c1x + c2x^2 + ...

// La funzione di astrazione è
// AF(c) = c0 + c1x + c2x^2 + ...
// dove
```

```
// ci = c.trms[i] se 0 <= i < c.trms.size
//     = 0, altrimenti
```

Nota bene: non è necessario fornire funzioni di astrazione per i tipi di record. Un tipo di record non fornisce alcuna astrazione sul suo *rep*; il suo *rep* è un insieme di campi e così sono i suoi oggetti astratti. Pertanto, la sua funzione di astrazione è sempre la mappa di identità.

5.5.2 - L'invariante di rappresentazione

In Java, il controllo di tipo garantisce che ogni volta che viene chiamato un metodo o un costruttore, il suo oggetto appartiene alla sua classe. Spesso, però, non tutti gli oggetti della classe sono rappresentazioni legittime di oggetti astratti.

Ad esempio, la rappresentazione di `IntSet` potrebbe potenzialmente includere oggetti il cui vettore `els` contenesse più di una voce con lo stesso valore intero. Tuttavia, abbiamo deciso che ogni elemento dell'insieme sarebbe stato inserito nel vettore esattamente una volta.

Pertanto, le rappresentazioni legittime di `IntSet` non contengono voci duplicate.

L'affermazione di una proprietà che tutti gli oggetti legittimi soddisfano è chiamata ***invariante di rappresentazione*** o *invariante di rep*. Un *invariante di rep* I è un predicato:

```
I: C --> boolean
```

Tale predicato è vero per gli oggetti legittimi.

Ad esempio, per `IntSet`, si potrebbe affermare il seguente invariante di rappresentazione:

```
java
// L'invariante di rappresentazione è
// c.els ≠ null &&
// per tutti gli interi i . c.els[i] è un Integer &&
// per tutti gli interi i, j . (0 ≤ i < j < c.els.size =>
//     c.els[i].intValue ≠ c.els[j].intValue)
```

Pertanto, I è falso se `els` contiene duplicati; inoltre, esclude una *rep* in cui `els` non si riferisce a un vettore, così come una *rep* in cui il vettore `els` contiene qualcosa di diverso da un `Integer`.

L'invariante di rappresentazione può anche essere dato in modo più informale:

```
java
// L'invariante di rappresentazione
// c.els ≠ null &&
// tutti gli elementi di c.els sono Integers &&
// non ci sono duplicati in c.els
```

Come secondo esempio, si consideri una rappresentazione alternativa di `IntSet` che consiste in

un array di 100 booleani più un vettore:

```
java
private boolean[100] els;
private Vector otherEls;
private int sz;
```

L'idea è che per un intero i nell'intervallo $0...99$, registriamo l'appartenenza all'insieme memorizzando true in `els[i]`. Gli interi al di fuori di questo intervallo vengono memorizzati in `otherEls`, come nella nostra precedente implementazione di `IntSet`. Poiché sarebbe costoso calcolare la dimensione dell'`IntSet` se dovessimo esaminare ogni parte dell'array `els`, memorizziamo anche la dimensione esplicitamente in `rep`. Questa rappresentazione è buona se quasi tutti i membri dell'insieme sono nell'intervallo $0...99$ e se ci aspettiamo che l'insieme abbia molti membri in questo intervallo. In caso contrario, lo spazio richiesto per l'array `els` sarà sprecato. Per questa rappresentazione si ha:

```
java
// La funzione di astrazione è:
// AF(c) = {c.otherEls[i].intValue | 0 <= i < c.otherEls.size}
//      +
//      {j | 0 <= j < 100 && c.els[j]}
```

In altre parole, l'insieme è l'unione degli elementi di `otherEls` e degli indici degli elementi veri di `els`. Inoltre, si ha:

```
java
// L'invarianza di rappresentazione è
// c.els != null && c.otherEls != null && c.els.size = 100 &&
// tutti gli elementi in c.otherEls sono Integers &&
// tutti gli elementi in c.otherEls non sono nel range da 0 a 99 &&
// non ci sono duplicati in c.otherEls &&
// c.sz = c.otherEls.size + (count of true entries in c.els)
```

Nota bene: la variabile di istanza `sz` di questa `rep` è ridondante: essa contiene informazioni che possono essere calcolate direttamente dalle altre variabili di istanza. Ogni volta che ci sono informazioni ridondanti nella `rep`, la relazione di queste informazioni con il resto della `rep` dovrebbe essere spiegata nell'invariante della `rep`.

A volte è conveniente utilizzare una funzione di aiuto nell'invariante `rep` o nella funzione di astrazione.

Ad esempio, l'ultima riga della precedente invariante `rep` può essere riscritta come segue:

```
java
// c.sz = c.otherEls.size + cnt(c.els,0)
// dove cnt(a,i) = if i >= a.size then 0
// else if a[i] then 1 + cnt(a, i+1)
```

```
// else cnt(a, i+1)
```

La funzione di aiuto `cnt` è definita da una relazione di ricorrenza.

L'implementazione di `Poly` presenta un'interessante invariante di ripetizione. Ricordiamo che abbiamo scelto di memorizzare i coefficienti solo fino al grado, senza zeri finali, tranne nel caso del polinomio zero. Pertanto, non ci aspettiamo di trovare uno zero nell'elemento alto della componente `trms`, a meno che la componente non abbia un solo elemento. Inoltre, questi array hanno sempre almeno un elemento. Inoltre, `deg` deve essere uno in meno della dimensione di `trms`.

Si ha quindi:

```
java
// L'invariante di rappresentazione
// c.trms != null && c.trms.length >= 1 && c.deg = c.trms.length-1
// && c.deg > 0 => c.trms[deg] != 0
```

Ricordiamo che l'implementazione dell'operazione `coeff` dipendeva dal fatto che la lunghezza dell'array fosse maggiore del grado del `Poly`; ora vediamo questo requisito espresso nell'invariante `rep`.

Nota bene: A volte tutti gli oggetti concreti sono rappresentazioni legali. Allora abbiamo semplicemente:

```
java
// L'invariante di rappresentazione è
// true
```

Questo è ciò che accade per i tipi di record: gli oggetti record vengono utilizzati accedendo direttamente ai loro campi. Questo significa che utilizzando il codice sarà in grado di **modificare i campi**, il che a sua volta significa che la classe che implementa il record non può vincolare in alcun modo la `rep`. Naturalmente, potrebbero esserci dei vincoli sul modo in cui vengono usati gli oggetti record che definiscono una relazione più forte tra i campi, ma questi vincoli sarebbero garantiti dal codice che usa gli oggetti record e si vedrebbero nell'invariante di `rep` per quel codice. Per esempio, l'invariante `rep` per l'implementazione del polinomio rado discussa nella includerebbe:

```
java
// per tutti gli elementi e di c.trms
// e è un Pair e e.exp >= 0 e e.coeff != 0
```

Non è necessario fornire gli invarianti di `rep` per i tipi di record, perché tutte queste classi hanno

esattamente lo stesso invariante di *rep*. Devono essere forniti per tutti gli altri tipi, anche per quelli per cui l'invariante è semplicemente vero. Fornire l'invariante può evitare che l'implementatore dipenda da un invariante più forte e non soddisfatto.

5.5.3 - Implementazione della AF e dell'IR

Oltre a fornire la funzione di astrazione e l'invariante *rep* come elementi del codice, è necessario fornire anche i **metodi per implementarli**. Questi metodi sono utili per trovare **errori nel codice**; inoltre, l'implementazione della funzione di astrazione può essere usata per fare output.

Il metodo `toString` viene utilizzato per implementare la funzione di astrazione. Il metodo che verifica l'invariante *rep* si chiama `repOk` e ha le seguenti specifiche:

```
public boolean repOk()  
    // EFFECTS: Ritorna true se l'invariante di rappresentazione è valido  
    // altrimenti restituisce false.
```

Il metodo è pubblico perché vogliamo che sia richiamabile da codice esterno alla sua classe. Ogni tipo dovrebbe fornire questo metodo, ma non è necessario fornire una specifica per esso, poiché la specifica è identica per ogni tipo.

Ad esempio, `repOk` nelle due classi viste finora:

```
java  
// per Poly:  
public boolean repOk() {  
    if (trms == null || deg != trms.length - 1 || trms.length == 0)  
        return true;  
    if (deg == 0) return true;  
    return trms[deg] != 0;  
}  
  
// per IntSet:  
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;  
    }  
}
```

Si noti l'uso dell'operatore `instanceof` in `repOk` per `IntSet` per verificare che l'elemento sia un intero.

Il metodo `repOk` viene utilizzato in due modi. I programmi di test possono chiamarlo per **verificare** se un'implementazione conserva l'invariante `rep`. Oppure si può usare il metodo all'interno delle **implementazioni di metodi e costruttori**. In questo caso, se l'invariante `rep` non è valido, si può lanciare `FailureException`. I costruttori lo richiamano prima di tornare, per assicurarsi che l'invariante `rep` sia valido sull'oggetto appena inizializzato. Inoltre, tutti i metodi che modificano l'invariante `rep` di oggetti vecchi o appena creati devono richiamarli prima di restituirli.

Ad esempio, in `Poly`, le routine `add`, `mul` e `minus` lo chiamerebbero, ma `sub` non ne ha bisogno, poiché non accede direttamente alle reps degli oggetti, e `coeff` non ne ha bisogno, poiché non modifica le reps. In `IntSet`, i mutatori `insert` e `remove` lo chiamerebbero.

Se le chiamate a `repOk` sono costose, possono essere disabilite quando il programma è in produzione.

Quindi, per ricapitolare:

- La **funzione di astrazione** spiega l'interpretazione del rappresentante. Mappano lo stato di ogni oggetto di rappresentazione legale all'oggetto astratto che si intende rappresentare. È implementata dal metodo `toString`.
- L'**invariante di rappresentazione** definisce tutte le assunzioni comuni che sono alla base delle implementazioni delle operazioni di un tipo. Definisce quali rappresentazioni sono legali, mappando ogni oggetto di rappresentazione a `true` (se la sua rappresentazione è legale) o a `false` (se la sua rappresentazione non è legale). È implementato dal metodo `repOk`.

5.5.4 - Discussione

Un invariante di `rep` è "invariante" perché è **sempre valido per le rep di oggetti astratti**; cioè, è valido ogni volta che un oggetto viene usato al di fuori della sua implementazione. Non è necessario che l'invariante `rep` sia sempre valido, poiché può essere violato durante l'esecuzione di una delle operazioni del tipo.

Ad esempio, il metodo `Poly mul` produce un componente `trms` con zero nell'elemento alto, ma l'elemento viene sovrascritto con un valore non nullo prima che `mul` ritorni.

Nota bene: L'invariante `rep` deve valere ogni volta che le operazioni ritornano ai loro chiamanti.

Esiste una relazione tra la funzione di astrazione e la rappresentazione invariante. La funzione di astrazione è interessante solo per le **rappresentazioni legali**, poiché solo queste rappresentano gli oggetti astratti. Pertanto, non è necessario definirla per le rappresentazioni illegali.

Ad esempio, sia `IntSet` che `Poly` hanno funzioni di astrazione che sono definite solo se le componenti `els` e `trms`, rispettivamente, sono non nulli; inoltre, la funzione di astrazione per `IntSet` ha senso solo se tutti gli elementi del `Vector` sono `Integer`.

C'è un problema che riguarda quanto dire in un invariante di *rep*. Un invariante di *rep* dovrebbe esprimere **tutti i vincoli da cui dipendono le operazioni**. Un buon modo di pensare è immaginare che le operazioni siano implementate da persone diverse che non possono parlare tra loro; l'invariante di *rep* deve contenere tutti i vincoli da cui dipendono i vari implementatori. Tuttavia, non è necessario dichiarare vincoli aggiuntivi.

Quando si implementa un'astrazione di dati, l'invariante *rep* è una delle prime cose a cui il programmatore pensa. **Deve essere scelto prima di implementare qualsiasi operazione**, altrimenti le implementazioni non lavoreranno insieme in modo armonioso. Per assicurarsi che venga compreso, l'invariante *rep* dovrebbe essere scritto e incluso come commento nel codice (oltre alla funzione di astrazione). La scrittura dell'invariante *rep* costringe l'implementatore ad articolare ciò che è noto e aumenta le probabilità che le operazioni vengano implementate correttamente. Tutte le operazioni devono essere implementate in modo tale da *preservare* l'invariante di rappresentazione.

Per esempio, supponiamo di implementare `insert` con:

```
java
public void insert(int x) {
    els.addElement(new Integer(x));
}
```

Questa implementazione può produrre un oggetto con elementi duplicati. Se sa che l'invariante *rep* vieta tali oggetti, questa implementazione è chiaramente scorretta.

L'invariante *rep* è utile anche per il lettore di un'implementazione.

Ad esempio, in un'implementazione alternativa di `IntSet`, avremmo potuto decidere di mantenere l'array *rep* ordinato. In questo caso, avremmo:

```
java
// L'invariante di rappresentazione è
//  c.els ≠ null && tutti gli elementi di c.els sono Integers
//  && per tutti i, j tale che 0 ≤ i < j < c.els.size
//      c.els[i].intValue < c.els[j].intValue
```

e le operazioni sarebbero state implementate in modo diverso rispetto a quanto fatto precedentemente.

Quindi, L'invariante *rep* spiega al lettore perché le operazioni sono implementate così come sono.

5.6 - Proprietà delle implementazioni dell'astrazione di dati

5.6.1 - Effetti collaterali benevoli

Un'astrazione mutabile deve avere un rappresentante mutabile, altrimenti non sarà possibile fornire la

mutabilità richiesta. Tuttavia, un'astrazione immutabile non deve necessariamente avere un rappresentante immutabile.

Nota bene: in Java è possibile avere un rappresentante immutabile dichiarando tutte le variabili di istanza come `final`.

Ad esempio, `Poly` è immutabile, ma hanno un rappresentante mutabile.

Un mutabile non è un problema, a patto che le modifiche apportate al rappresentante non possano essere osservati dagli utenti dell'astrazione.

Ad esempio, a volte è utile inizializzare un oggetto modificando in modo incrementale la sua *rep*, anche se una volta che l'oggetto è completamente inizializzato, la sua *rep* non viene più modificata. Questo è il modo in cui vengono create i polinomi in alcuni metodi `Poly`.

Quindi, la mutabilità è anche utile per **effetti collaterali benevoli**, ossia modifiche non visibili all'esterno dall'implementazione.

Ad esempio, supponiamo che i numeri razionali siano rappresentati come una coppia di numeri interi:

```
java
int num, denom;
```

La funzione di astrazione è

```
java
// Un tipico razionale è n/d

// La funzione di astrazione è
// AF(c) = c.num/c.denom
```

Data questa rappresentazione, si devono fare diverse scelte:

- cosa fare con un denominatore zero,
- come memorizzare i razionali negativi
- se mantenere o meno il razionale in forma ridotta (cioè con il numeratore e il denominatore ridotti in modo che non ci siano termini comuni).

Supponiamo di scegliere di escludere i denominatori nulli, di rappresentare i razionali negativi con numeratori negativi e di *non* mantenere il razionale in forma ridotta (per velocizzare operazioni come la moltiplicazione). Si ha quindi

```
java
// L'invariante di rappresentazione è
// c.denom > 0
```

Tuttavia, per verificare se due razionali sono uguali, è utile calcolare le forme ridotte; esse possono essere calcolate utilizzando la seguente procedura `gcd`:


```

java
static public int gcd(int n, int d) throws NonPositiveException
    // EFFECTS: Se n o d non sono positive lancia NonPositiveException
    // altrimenti ritorna il comune divisore più grande di n e di d

```

L'implementazione di equals:

```

java
public class rat {
    private int num;
    private int denom;

    public boolean equals(rat r) {
        if (r == null) return false;
        if (num == 0) return r.num == 0;
        if (r.num == 0) return false;
        reduce();
        r.reduce();
        return (num == r.num && denom == r.denom);
    }

    private void reduce() {
        // REQUIRES: This.num != 0
        // MODIFIES: This
        // EFFECTS: Cambia this nella sua forma ridotta
        int temp = num;
        if (num < 0) temp = -num;
        int g = Num.gcd(temp,denom);
        num = num/g;
        denom = denom/g;
    }
}

```

Una volta calcolate, le forme ridotte vengono memorizzate nella *rep*, per velocizzare il successivo test di uguaglianza. La modifica della ripetizione eseguita con il metodo `equals` è un effetto collaterale benefico.

Tali effetti collaterali vengono spesso eseguiti per **motivi di efficienza**. Sono possibili quando la funzione di astrazione è molti a uno, poiché molti oggetti *rep* rappresentano un particolare oggetto astratto. A volte è utile all'interno di un'implementazione passare da uno di questi oggetti *rep* a un altro. Questo passaggio è sicuro, poiché la *rep* continua a rappresentare lo stesso oggetto astratto.

5.6.2 - Esporre il rappresentante

Una questione chiave nell'implementazione delle astrazioni di dati è ottenere la capacità di fare ragionamenti locali: vogliamo essere in grado di garantire che una classe sia corretta semplicemente esaminando il codice di quella classe. Il **ragionamento locale** è valido solo se le rappresentazioni degli oggetti astratti **non possono essere modificate** al di fuori della loro implementazione. Se il ragionamento locale non è supportato, si dice che l'implementazione **espone il rappresentante**. Esporre il rappresentante significa che l'implementazione rende accessibili al codice esterno alla classe le componenti mutabili del rappresentante.

Un modo per esporre il rappresentante è quello di avere dichiarazioni di variabili di istanza che non siano dichiarate come `private`. Tuttavia, anche se tutte le variabili di istanza sono private, è comunque possibile esporre il rappresentante!

Ad esempio, supponiamo che `IntSet` abbia un metodo `allEls` con le seguenti specifiche:

```
java
public Vector allEls()
    // EFFECTS: Ritorna un array contenente gli elementi di questo,
    // ciascuno esattamente una volta, in un ordine arbitrario.
```

e supponiamo che questo metodo venga implementato come segue:

```
java
public Vector allEls() {
    return els;
}
```

Questa implementazione consentirebbe agli utenti di `IntSet` di accedere direttamente al componente `els`; poiché questo componente è mutabile, gli utenti possono modificarlo. Per evitare questo problema, l'implementazione di `allEls` deve restituire una copia del componente `els`.

L'esposizione della `rep` è un errore di implementazione. Può accadere sia perché un metodo restituisce un oggetto mutabile nella `rep`, come discusso in precedenza, sia perché un costruttore o un metodo rende un oggetto argomento mutabile parte della `rep`.

Ad esempio, supponiamo che `IntSet` abbia il seguente costruttore:

```
java
public IntSet(Vector elms) throws NullPointerException
    // EFFECTS: Se elms è null lancia NullPointerException altrimenti
    //  inizializza this in modo che contenga come elementi tutti
    //  gli ints in elms
```

E supponiamo che l'implementazione sia:

```
java
public IntSet(Vector elms) throws NullPointerException {
    if (elms == null)
        throw new NullPointerException ("IntSet 1 argument constructor");
    els = elms;
}
```

Anche in questo caso, si verifica un errore di implementazione che comporta l'esposizione del rappresentante.

5.7 - Ragionare sulle astrazioni di dati

Ogni volta che si scrive un programma, **si ragiona in modo informale sulla sua correttezza**. Questo ragionamento è così elementare che potreste non essere consapevoli di farlo! Inoltre, spesso si vuole convincere gli altri della correttezza del proprio codice, ad esempio nell'ambito di un'ispezione del codice. Questo processo di "convincimento" comporta anche un'argomentazione informale sulla correttezza. Infine, quando si legge il codice di qualcun altro per determinare se è corretto, si procede anche a un'argomentazione informale di correttezza.

Ragionare sulla correttezza delle procedure autonome è relativamente semplice: si assume che la preconditione sia valida e si esamina il codice per convincersi che la procedura faccia ciò che la sua clausola sugli effetti richiede. Ragionare sulle implementazioni di astrazioni di dati è un po' più complicato, perché bisogna **considerare l'intera classe**. Inoltre, si deve ragionare su codice scritto a livello concreto (cioè che manipola la ripetizione), pur convincendosi che soddisfa la specifica, che è scritta in termini di oggetti astratti.

In primo luogo si discute come dimostrare che un'implementazione **preservi l'invariante di rappresentazione**, cioè assicura che l'invariante sia vero per un oggetto ogni volta che viene utilizzato al di fuori della sua classe. Si discute poi come ragionare sul fatto che le **operazioni facciano la cosa giusta**. Si discute anche di come ragionare sulle proprietà di un'astrazione di dati **dimostrando che certi invarianti astratti sono validi**.

5.7.1 - Preservare l'invariante di rappresentazione

Per dimostrare che un tipo è implementato correttamente, dobbiamo dimostrare che l'invariante *rep* è valido per tutti gli oggetti della classe. Lo facciamo nel modo seguente. Innanzitutto, dimostriamo che **l'invariante è valido per gli oggetti restituiti dai costruttori**. Per i metodi, possiamo assumere che quando vengono chiamati l'invariante vale per `this` e per tutti gli oggetti del tipo; dobbiamo dimostrare che vale quando il **metodo ritorna per `this`, per tutti gli argomenti del tipo** e anche per **gli oggetti restituiti del tipo**.

Ad esempio, l'implementazione di `IntSet` ha l'invariante:

```
java
// c.els != null &&
// per tutti gli interi i . c.els[i] è un Integer &&
// per tutti gli interi i, j . (0 <= i < j < c.els.size =>
//  c.els[i].intValue != c.els[j].intValue)
```

- Il costruttore `IntSet` stabilisce questa invariante perché il vettore appena creato è vuoto.
- Il metodo `isIn` lo preserva perché possiamo assumere che l'invariante sia valido per `this` quando viene chiamato `isIn` e `isIn` non modifica `this`; lo stesso vale per `size` e `choose` e per il metodo privato `getIndex`.
- Il metodo `insert` conserva l'invariante perché sono soddisfatte le seguenti condizioni:
 - ♦ L'invariante è valida al momento della chiamata
 - ♦ La chiamata a `getIndex` da parte di `insert` preserva l'invariante
 - ♦ `insert` aggiunge `x` a `this` solo se `x` non è già presente in `this` (cioè, se `getIndex(x)` restituisce `-1`); pertanto, poiché `this` soddisfa l'invariante al momento della chiamata, continua a soddisfare l'invariante anche dopo la chiamata.

Come secondo esempio, si consideri l'implementazione di `Poly` e si ricordi che l'invariante è:

```
java
// c.trms != null && c.trms.length >= 0 && c.deg = c.trms.length - 1
// && c.deg > 0 => c.trms[deg] != 0
```

- Il costruttore `Poly` che produce il polinomio zero preserva la variante in quanto crea un array a un elemento; l'altro costruttore `Poly` preserva l'invariante perché verifica esplicitamente la presenza del polinomio zero.
- L'operazione `mul` preserva l'invariante perché sono soddisfatte le seguenti condizioni:
 - ♦ L'invariante vale per `this` al momento della chiamata; vale anche per `q` se `q` non è nullo.
 - ♦ Se `this` o `q` sono lo zero `Poly`, ciò viene riconosciuto e viene costruita la *rep* corretta
 - ♦ Altrimenti, né `this` né `q` contengono uno zero nel loro termine alto; pertanto, il termine alto dell'array `trms` nel `Poly` restituito, che contiene il prodotto dei termini alti di `trms` e `q.trms`, non può essere zero.

Questo ragionamento è chiamato **induzione del tipo di dati**. L'induzione riguarda il numero di invocazioni di procedure utilizzate per produrre il valore attuale dell'oggetto. Il primo passo dell'induzione consiste nello stabilire la proprietà dei costruttori; il passo dell'induzione stabilisce la proprietà dei metodi.

5.7.2 - Ragionare sulle operazioni

Per convincersi che la propria implementazione è corretta occorre non solo dimostrare che le **operazioni conservano l'invariante *rep***, ma è anche necessario dimostrare che **ogni operazione fa ciò che deve fare**.

La difficoltà è che le specifiche sono scritte in termini di oggetti astratti, ma l'implementazione manipola rappresentazioni concrete. Occorre quindi un modo per mettere in relazione le due cose. Per farlo, si utilizza la funzione di astrazione.

Per esempio, supponiamo di voler sostenere che l'implementazione di `IntSet` sia corretta. Si tratterebbe di sostenere che ogni operazione è implementata correttamente:

- **Il costruttore**. Il costruttore `IntSet` restituisce un oggetto la cui componente `els` è un vettore vuoto. Questo è corretto perché la funzione di astrazione mappa il vettore vuoto nell'insieme vuoto.
- **Il metodo `size`**. Quando questo metodo viene chiamato, sappiamo che la dimensione del vettore `els` è la cardinalità dell'insieme, perché la funzione di astrazione mappa gli elementi del vettore negli elementi dell'insieme e perché l'invariante *rep*, che si può assumere valido quando viene chiamato `size`, assicura che non ci siano duplicati in *els*. Pertanto, la restituzione di questa dimensione è corretta.
- **Il metodo `remove`**. Questo metodo controlla innanzitutto se l'elemento da rimuovere è presente nel vettore e, in caso contrario, lo restituisce. Questo è corretto perché se l'elemento non è nel vettore, non è nell'insieme (a causa del modo in cui la funzione di astrazione mappa il vettore nell'insieme) e quindi, quando `remove` ritorna, `this.post` mappa a `this - { x }`. Altrimenti, il metodo rimuove l'elemento dal vettore, e anche in questo caso otteniamo il risultato giusto, perché l'invariante *rep* garantisce che non ci siano duplicati in `els`.

Questo processo continuerà fino a quando non saranno state considerate tutte le operazioni.

Nota bene: in queste prove ci avvaliamo dell'invariante *rep*, cioè siamo in grado di assumere che esso sia valido all'ingresso.

Un aspetto importante di queste prove è che siamo in grado di **ragionare su ogni operazione in modo indipendente**, il che è possibile grazie all'invariante *rep*. Esso cattura le assunzioni comuni tra le operazioni e, in questo modo, si sostituisce a tutte le altre operazioni quando consideriamo la prova di una particolare operazione. Naturalmente, questo ragionamento è valido solo se tutte le operazioni conservano l'invariante *rep*, poiché è questo che permette di prendere il posto delle altre operazioni nel processo di ragionamento.

5.7.3 - Ragionare a livello astratto

È anche utile ragionare su un'astrazione di dati a livello astratto. In questo caso, il ragionamento si basa solo sulla specifica del tipo; possiamo ignorare la sua implementazione.

Un tipo di proprietà che è utile dimostrare è l'**invariante astratto**, che è l'analogo astratto dell'invariante rep.

Per esempio, ci siamo basati su invarianti astratti per vettori e array nel nostro ragionamento sulla correttezza delle implementazioni di `IntSet` e `Poly`. Per entrambi i vettori e gli array, abbiamo assunto che la loro dimensione fosse maggiore o uguale a zero e, inoltre, che tutti gli indici maggiori o uguali a zero e minori della dimensione fossero nei limiti.

Esistono invarianti astratti simili per gli insiemi e i polinomi. Ad esempio, la dimensione di un `IntSet` è sempre maggiore o uguale a zero. Questa proprietà può essere stabilita come segue:

- È chiaro che vale per il costruttore, poiché restituisce un nuovo `IntSet` vuoto.
- È valido per `insert`, poiché aumenta solo la dimensione di `IntSet`.
- Questo vale per `remove`, poiché rimuove un elemento dall'insieme solo se l'elemento era presente nell'insieme al momento della chiamata.

Si noti che in questa prova **ignoriamo completamente gli osservatori**. Poiché non modificano i loro oggetti (in modo che gli utenti possano notarli), non possono influenzare la proprietà.

Si noti che stiamo ragionando a livello astratto, non a livello di implementazione. Non ci preoccupiamo di come vengono implementati gli `IntSet`. Lavoriamo invece direttamente con le specifiche di `IntSet`. Lavorare a livello astratto **semplifica notevolmente il ragionamento**.

5.8 - Problemi di progettazione

5.8.1 - Mutabilità

Le astrazioni di dati possono essere mutevoli, con oggetti i cui valori possono cambiare, o immutabili. È necessario prestare attenzione a questo aspetto di un tipo.

In generale, un tipo dovrebbe essere **immutabile** se i suoi oggetti hanno naturalmente valori immutabili.

È il caso, per esempio, di oggetti matematici come numeri interi, `Polys` e numeri complessi. Un tipo dovrebbe essere **mutevole** se sta modellando qualcosa del mondo reale, dove è naturale che i valori degli oggetti cambino nel tempo.

Ad esempio, un'automobile in un sistema di simulazione può essere in esecuzione o fermo, e contenere o meno passeggeri. Allo stesso modo, un tipo che modella la memoria, come un array o un insieme, è probabile che sia mutabile. Tuttavia, in questi casi si potrebbe preferire l'uso di un tipo immutabile, per la maggiore sicurezza che l'immutabilità offre o perché l'immutabilità può consentire la condivisione di sottoparti.

Nel decidere la mutabilità, a volte è necessario fare un **compromesso tra efficienza e sicurezza**. Le

astrazioni immutabili sono più sicure di quelle mutabili, perché non sorgono problemi se i loro oggetti sono condivisi. Tuttavia, per le astrazioni immutabili è possibile che vengano creati e scartati frequentemente nuovi oggetti, il che significa che il lavoro di gestione dello *storage* (ad esempio, la *garbage collection*) viene svolto con maggiore frequenza.

Ad esempio, rappresentare un insieme come una lista non è probabilmente una buona scelta se si utilizzano frequentemente le operazioni di inserimento e rimozione.

In ogni caso, si noti che **la mutabilità o l'immutabilità è una proprietà del tipo** e non della sua implementazione. Un'implementazione deve semplicemente supportare questo aspetto del comportamento della sua astrazione.

5.8.2 - Categorie di operazioni

Le operazioni di un'astrazione di dati rientrano in quattro categorie:

1. **Creatori:** Queste operazioni **creano “da “zero” oggetti del loro tipo**, senza prendere alcun oggetto del loro tipo come input. Tutti i creatori sono costruttori. La maggior parte dei costruttori sono creatori. Ma a volte i costruttori prendono argomenti del loro tipo e questi non sono creatori.
2. **Produttori:** Queste operazioni **prendono in ingresso** oggetti del loro tipo e **creano** altri oggetti del loro tipo. Possono essere costruttori o metodi.
Ad esempio, `add` e `mul` sono produttori di `Poly`
3. **Mutatori:** Sono metodi che **modificano gli oggetti** del loro tipo. È chiaro che solo i tipi mutabili possono avere mutatori.
Ad esempio, `insert` e `remove` sono mutatori per `IntSet`.
4. **Osservatori:** Sono metodi che prendono in ingresso oggetti del loro tipo e restituiscono risultati di altri tipi. Vengono utilizzati per ottenere **informazioni sugli oggetti**.
Esempi sono `size`, `isIn` e `choose` per `IntSet`, `coeff` e `degree` per le `Poly`

Ricorda: i creatori di solito producono alcuni oggetti, ma **non tutti**.

Per esempio, i creatori `Poly` (i due costruttori) producono solo polinomi a termine singolo, mentre il costruttore `IntSet` produce solo l'insieme vuoto.

Gli altri oggetti sono prodotti da **produttori** o **mutatori**.

Ad esempio, il produttore `add` può essere utilizzato per ottenere polinomi con più di un termine, mentre il mutatore `insert` può essere utilizzato per ottenere insiemi contenenti molti elementi.

Nota bene: i mutatori hanno lo stesso ruolo nei tipi mutabili che i produttori hanno in quelli immutabili. Un tipo mutabile può avere sia produttori che mutatori.

Per esempio, se `IntSet` avesse un metodo `clone`, questo metodo sarebbe un produttore. A volte gli osservatori sono combinati con produttori o mutatori.

Per esempio, `IntSet` potrebbe avere un metodo `chooseAndRemove` che restituisce l'elemento scelto e lo rimuove anche dall'insieme.

5.8.3 - Adeguatezza

Un tipo di dati è **adeguato** se fornisce un **numero sufficiente di operazioni** in modo che tutto ciò che gli utenti devono fare con i suoi oggetti possa essere fatto in modo conveniente e con ragionevole efficienza. Non è possibile dare una definizione precisa di adeguatezza, anche se ci sono dei limiti al numero di operazioni che un tipo può avere ed essere ancora utile.

Per esempio, se forniamo solo il costruttore `IntSet` e i metodi `insert` e `remove`, i programmi non possono scoprire nulla sugli elementi dell'insieme (perché non ci sono osservatori). D'altro canto, se a queste tre operazioni aggiungiamo solo il metodo `size`, possiamo conoscere gli elementi dell'insieme (ad esempio, si potrebbe verificare l'appartenenza cancellando l'intero e vedendo se la dimensione è cambiata), ma il tipo sarebbe costoso e scomodo da usare.

Una nozione molto rudimentale di adeguatezza può essere ottenuta considerando le categorie di operazioni. In generale, un'astrazione di dati deve avere operazioni appartenenti ad almeno **tre delle quattro categorie** discusse nella sezione precedente. Deve avere creatori, osservatori e produttori (se è immutabile) o mutatori (se è mutabile). Inoltre, il tipo deve essere **completamente popolato**. Ciò significa che tra i suoi creatori, mutatori e produttori deve essere possibile ottenere ogni possibile stato dell'oggetto astratto.

Tuttavia, la nozione di adeguatezza deve tenere conto anche del **contesto d'uso**: un tipo deve avere un insieme di operazioni sufficientemente ricco per gli usi previsti. Se il tipo deve essere usato in un contesto limitato, come un singolo pacchetto, allora è sufficiente fornire le operazioni necessarie per quel contesto. Se il tipo è destinato a un uso generale, è auspicabile un insieme ricco di operazioni. Per decidere se un'astrazione di dati ha un numero sufficiente di operazioni, occorre identificare tutto ciò che gli utenti potrebbero ragionevolmente aspettarsi di fare. Successivamente, si deve pensare a come queste operazioni possono essere eseguite con l'insieme di operazioni fornito. Se qualcosa sembra troppo costoso o troppo macchinoso (o entrambe le cose), si può verificare se l'aggiunta di un'operazione possa essere utile. A volte è possibile ottenere un miglioramento sostanziale delle prestazioni semplicemente avendo accesso alla rappresentazione.

Ad esempio, si potrebbe eliminare l'operazione `isIn` per gli insiemi `Int`, perché questa operazione può essere implementata al di fuori del tipo utilizzando le altre operazioni. Tuttavia, la verifica dell'appartenenza a un insieme è un uso comune e sarà più veloce se eseguita all'interno dell'implementazione. Pertanto, `IntSet` dovrebbe fornire questa operazione.

Un tipo può anche avere un numero eccessivo di operazioni. Quando si considera l'aggiunta di operazioni, occorre considerare come si adattano allo scopo dell'astrazione dei dati.

Per esempio, un'astrazione di memorizzazione come `Vector` o `IntSet` dovrebbe includere operazioni per accedere e modificare la memorizzazione, ma non operazioni non correlate a questo scopo, come un metodo di ordinamento o un metodo per calcolare la somma degli elementi del vettore o dell'insieme.

Un numero eccessivo di operazioni rende l'astrazione più difficile da capire. Inoltre, l'implementazione è più difficile e anche la manutenzione, perché se l'implementazione cambia, viene coinvolto più codice. La desiderabilità di operazioni aggiuntive deve essere bilanciata da questi fattori. Se il tipo è adeguato, le sue operazioni possono essere aumentate da procedure autonome che sono al di fuori dell'implementazione del tipo (cioè, metodi statici di qualche altra classe).

5.9 - Località e modificabilità

I vantaggi della localizzazione e della modificabilità si applicano sia alle astrazioni di dati che alle procedure. Tuttavia, questi vantaggi possono essere ottenuti solo se si dispone di un'astrazione tramite specifica.

La **località** (la capacità di ragionare su un modulo guardando solo il suo codice e non qualsiasi altro codice) richiede che una rappresentazione sia **modificabile solo all'interno dell'implementazione** del suo tipo. Se le modifiche possono avvenire altrove, non possiamo stabilire la correttezza dell'implementazione solo esaminando il suo codice.

Per esempio, non possiamo garantire localmente che l'invariante `rep` sia valido e non possiamo usare l'induzione dei tipi di dati con sicurezza.

La **modificabilità** (la capacità di reimplementare un'astrazione senza dover reimplementare altro codice) richiede ancora di più della localizzazione: tutti gli accessi a una rappresentazione, anche ai componenti immutabili, devono avvenire **all'interno dell'implementazione** del suo tipo. Se l'accesso avviene in qualche altro modulo, non possiamo sostituire l'implementazione senza influenzare l'altro modulo. Per questo motivo, tutte le variabili di istanza devono essere dichiarate `private`. Per questo motivo è fondamentale che l'accesso alla rappresentazione sia limitato ai soli dell'implementazione del tipo. È auspicabile che il linguaggio di programmazione aiuti in questo senso, in modo che l'accesso limitato sia garantito a condizione che l'implementatore non esponga la ripetizione. Altrimenti, l'accesso limitato è un'altra proprietà che deve essere dimostrata sui programmi. Java fornisce un supporto per l'accesso limitato attraverso i suoi meccanismi di incapsulamento.

Quindi, ricapitolando:

- L'implementazione di un'astrazione di dati garantisce la **localizzazione** se il codice che la utilizza non può modificare i componenti del `rep`; in altre parole, **non deve esporre il rep**.
- Un'implementazione dell'astrazione dei dati fornisce la **modificabilità** se, inoltre, non c'è modo per il codice utilizzato di accedere a qualsiasi parte del rappresentante.

6 - Astrazione dell'iterazione

Un uso comune di una collezione è quello di eseguire un'azione sui suoi elementi. Per questo motivo, è necessario un metodo per accedere a tutti gli elementi. Questo metodo deve essere efficiente in termini di spazio e tempo, comodo da usare e non distruttivo per la collezione. Inoltre, deve supportare l'astrazione per specificazione. Gli iteratori risolvono questi problemi.

Gli iteratori sono una generalizzazione dei meccanismi di iterazione disponibili nella maggior parte dei linguaggi di programmazione. Permettono di iterare su tipi arbitrari di dati in modo comodo ed efficiente.

Ad esempio, un uso ovvio di un insieme è quello di seguire un'azione per ciascuno dei suoi elementi:

```
for tutti gli elemento dell'insieme
    fare un'azione
```

Un ciclo di questo tipo può attraversare completamente l'insieme, ad esempio per sommare tutti gli elementi di un insieme. Oppure può cercare un elemento che soddisfi un criterio, nel qual caso il ciclo può interrompersi non appena viene trovato l'elemento desiderato.

Gli insiemi `IntSet` così come sono stati definiti finora non forniscono un modo conveniente per eseguire tali cicli.

Per esempio, supponiamo di voler calcolare la somma degli elementi di un `IntSet`:

```
java
public static int setSum(IntSet s) throws NullPointerException
    // EFFECTS: Se s è null lancia NullPointerException altrimenti
    //  ritorna la somma degli elemento di s
```

L'implementazione di `setSum` è:

```
java
public static int setSum(IntSet s) throws NullPointerException {
    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]);
    }
    // inserimento degli elemento di s
    for (int i = 0; i < a.length; i++) s.insert(a[i]);
}
```

```
    return sum;
}
```

L'implementazione di `setSum` illustra i due principali difetti della nostra astrazione `IntSet` :

1. Per scorrere tutti gli elementi, cancelliamo ogni elemento restituito da `choose`, in modo che non venga più scelto. Pertanto, a ogni iterazione devono essere richiamate due operazioni, `choose` e `remove`. Questa inefficienza potrebbe essere evitata facendo in modo che `choose` rimuova l'elemento scelto.
2. L'iterazione su un `IntSet` lo distrugge rimuovendo tutti i suoi elementi. Tale distruzione può essere accettabile a volte, ma non può essere soddisfacente in generale. Sebbene sia possibile raccogliere gli elementi rimossi e reinserirli in un secondo momento, questo approccio è goffo e inefficiente.

Se `setSum` fosse un'operazione di `IntSet`, potremmo implementarla in modo efficiente manipolando i rappresentanti di `IntSet`. Tuttavia, `setSum` **non ha senso come operazione** di `IntSet`; sembra periferica rispetto al concetto di insieme. Inoltre, anche se potessimo giustificare l'operazione, che dire di altre procedure simili che potremmo desiderare? Deve esserci un modo per implementare tali procedure in modo efficiente al di fuori del tipo.

Per supportare adeguatamente l'iterazione, è necessario **accedere a tutti gli elementi** di una collezione in modo efficiente e senza distruggere la collezione

Come si può fare per gli `IntSet`? Una possibilità è quella di fornire un metodo `members` :

```
java
public int[] members()
    // EFFECTS: Ritorna un array contenente gli elementi di questo,
    // ciascuno esattamente una volta, in un ordine arbitrario.
```

Data questa operazione, possiamo implementare `setSum` :

```
java
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;
}
```

Poiché i `members` non modificano il loro argomento, non è più necessario ricostruire l'`IntSet` dopo l'iterazione. Sebbene `members` facilitino l'uso degli `IntSet`, è inefficiente, soprattutto se l'`IntSet` è grande. In primo luogo, abbiamo due strutture di dati: l'`IntSet` stesso e l'array e, se

l'insieme è grande, lo è anche l'array. In secondo luogo, nel caso di un ciclo di ricerca, probabilmente abbiamo fatto troppo lavoro, poiché non è necessario che il ciclo esamini tutti gli elementi dell'insieme ricercato. Ad esempio, se stessimo cercando un elemento negativo in un `IntSet`, potremmo fermarci non appena incontriamo il primo elemento negativo. Tuttavia, dobbiamo elaborare l'intero insieme per costruire l'array.

Un'alternativa a `members` è un'operazione che restituisce semplicemente il vettore rappresentante. Tuttavia, questa soluzione è pessima, poiché distrugge l'astrazione esponendo il vettore *rep*.

Un'altra possibilità è quella di modificare l'astrazione `IntSet` per includere la nozione di indicizzazione. Tuttavia, un `IndexedSet` è un'astrazione più complicata di `IntSet` e la complessità aggiunta non sembra intrinseca alla nozione di insieme.

È necessario un meccanismo generale di iterazione che sia conveniente ed efficiente e che preservi l'astrazione. Gli *iteratori* forniscono il supporto necessario.

Un **iteratore** è un tipo speciale di procedura che fa sì che gli **elementi** su cui vogliamo iterare vengano **prodotti in modo incrementale**. Gli elementi prodotti possono essere utilizzati in altri moduli che specificano le azioni da eseguire per ciascun elemento. Il codice utilizzato conterrà una sorta di struttura ad anello:

```
for each elemento di risultato "i" prodotto dall'iteratore A
    eseguire una qualche azione su "i"
```

Quindi, ogni iterazione del ciclo produce un nuovo elemento, che viene poi trattato dal corpo del ciclo.

Nota bene: L'iteratore è responsabile della **produzione degli elementi**, mentre il codice contenente il ciclo definisce l'azione da eseguire su di essi. L'iteratore può essere utilizzato in diversi moduli che eseguono azioni diverse sugli elementi e può essere implementato in modi diversi senza influire su questi moduli.

Poiché l'iteratore fa sì che gli elementi vengano prodotti uno alla volta, **evita i problemi di spazio e di tempo** discussi in precedenza. Non è necessario costruire una struttura dati potenzialmente grande per contenere gli elementi. Inoltre, se il codice in uso esegue un ciclo di ricerca, l'iteratore può essere interrotto non appena viene trovato l'elemento di interesse.

Come già detto, gli iteratori sono una generalizzazione dei meccanismi di iterazione disponibili nella maggior parte dei linguaggi di programmazione. Oltre a una qualche forma di ciclo `while`, i linguaggi di programmazione forniscono in genere un ciclo `for` per l'iterazione sui numeri interi. Tale iterazione è utile con gli array, che sono indicizzati, ma non si adatta bene a collezioni non indicizzate come `IntSet`. Gli iteratori forniscono una comoda iterazione anche su collezioni non indicizzate.

6.1 - Iterazione in Java

Java non fornisce un supporto diretto per l'astrazione dell'iterazione. Al contrario, l'iterazione viene fornita da un tipo speciale di procedura, che verrà chiamata **iteratore**. Alcuni iteratori sono metodi di astrazioni di dati e un'astrazione di dati può fornire diversi metodi di iteratori. Inoltre, possono esistere iteratori indipendenti.

Un iteratore restituisce un tipo speciale di oggetto di dati chiamato **generatore**. Un generatore **tiene traccia dello stato** di un'iterazione nel suo *rep*. Ha due metodi:

- `hasNext`: usato per determinare se rimangono altri elementi da produrre
- `next`: usato per ottenere l'elemento successivo e far avanzare lo stato dell'oggetto generatore per registrare il ritorno di quell'elemento.

Tutti i generatori appartengono a tipi che sono sottotipi dell'interfaccia `Iterator`, definita dal pacchetto `java.util`:

```
public interface Iterator {  
  
    public boolean hasNext();  
    // EFFECTS: Ritorna true se ci sono più elementi da produrre  
    // altrimenti restituisce false  
  
    public Object next() throws NoSuchElementException;  
    // MODIFIES: this  
    // EFFECTS: Se ci sono più risultati da restituire, restituisce il  
    // risultato successivo e modifica lo stato di this per registrare  
    // il rendimento. Altrimenti, lancia NoSuchElementException  
  
}
```

La specifica fornisce una descrizione generica di tutti i tipi di generatore; tutti questi tipi hanno oggetti con i due metodi e il comportamento indicato. `NoSuchElementException` è un'eccezione non controllata, perché si prevede che la maggior parte degli usi di un generatore eviterà di sollevare l'eccezione.

Un esempio di come si utilizza un'iteratore:

```
java  
// loop controllato da hasNext  
Iterator g = primesLT100();  
while (g.hasNext()) {  
    int x = ((Integer) g.next()).intValue();  
    // utilizzo di x  
}  
  
// loop controllato dall'eccezione  
Iterator g = primesLT100();
```

```

try {
    while (true) {
        int x = ((Integer) g.next()).intValue();
        // utilizzo di x
    }
}
catch (NoSuchElementException e){}

```

In questo esempio, l'iteratore `primesLT100` restituisce un generatore che produrrà tutti i numeri primi inferiori a *100*. Il generatore viene tipicamente utilizzato in un ciclo `while`. Il corpo del ciclo utilizza il metodo `next` per ottenere il valore successivo prodotto dall'iterazione. O il ciclo è controllato dal metodo `hasNext` oppure il ciclo può essere terminato quando il metodo `next` lancia un'eccezione.

6.2 - Specificare gli iteratori

La specifica di un iteratore spiega l'intera iterazione: come l'iteratore usa i suoi argomenti per produrre un generatore e il comportamento del generatore. La specifica fornita precedentemente spiega cosa fanno i metodi del generatore, ma è generica: non spiega esattamente cosa fa un particolare generatore. Le informazioni mancanti sono state raccolte nella specifica dell'iteratore

Un esempio di specifiche di iteratori:

```

java
public class Poly {
    // come prima più:

    public Iterator terms()
    // EFFECTS Ritorna un generatore che produce gli esponenti dei termini
    // non nulli di this (come Integers) fino al grado, in ordine
    // crescente di esponente.
}

public class IntSet {
    // come prima più:

    public Iterator elements()
    // EFFECTS: Ritorna un generatore che produce tutti gli elementi di
    // "this" (come interi), ciascuno esattamente una volta, in ordine
    // arbitrario.
    // REQUIRES: this non deve essere modificato mentre il generatore è
    // in uso.
}

```

- L'iteratore `terms` è un metodo di `Poly` che consente di iterare attraverso i termini di `Poly`. La specifica spiega che il generatore restituito consente l'iterazione su `this`, producendo tutti gli esponenti della sua `Poly` per i termini non nulli fino al grado. Si noti che la specifica indica il tipo di oggetto (`Integer`) che sarà effettivamente prodotto dal generatore.
- L'iteratore `elements` sostituisce il metodo `choose` descritto in precedenza. Due punti sono interessanti:
 1. Si noti che la specifica degli elementi include un requisito per il codice che utilizza il generatore. Non è chiaro cosa farebbe il generatore se l'insieme viene modificato durante l'utilizzo del generatore. Pertanto, escludiamo tali modifiche nella specifica di `elements`. Quasi sempre un generatore su un oggetto mutabile avrà questo requisito. Come di consueto, dichiariamo il requisito in una clausola *requires*, ma poiché si tratta di un requisito sull'uso del generatore, piuttosto che sulla chiamata all'iteratore, collochiamo la clausola *requires* alla fine della specifica. Normalmente, una clausola *requires* è la prima parte di una specifica. In effetti, un iteratore potrebbe avere **due clausole *requires***: una che **esclude alcuni argomenti** e l'altra che **indica i vincoli** sull'uso del generatore restituito.
 2. A differenza del metodo `choose`, l'iteratore di elementi **non lancia alcuna eccezione**. È tipico che l'uso degli iteratori elimini i problemi associati a certi argomenti (come l'insieme vuoto) che si presenterebbero per procedure correlate come `choose`.

Nota bene: Sebbene entrambe queste astrazioni di dati forniscano un solo iteratore, un'astrazione di dati può avere *molte iteratori*. Inoltre, né i `terms` né gli `elements` modificano nulla: l'iteratore non modifica `this` e nemmeno il generatore che lo restituisce. Se c'è una modifica, la specifica dell'iteratore deve spiegare di cosa si tratta e se è l'iteratore o il generatore a effettuare la modifica.

Oltre ai metodi iteratori, è possibile avere anche iteratori autonomi; saranno metodi statici.

Ad esempio:

```
java
public class Num {

    public static Iterator allPrimes()
        // EFFECTS: Restituisce un generatore che produce tutti i numeri
        // primi (come Integers), ciascuno esattamente una volta, in
        // ordine crescente.

}
```

Il generatore restituito da `allPrimes` continuerà a produrre risultati senza alcun limite, quindi dovrà essere utilizzato in un ciclo che limiti l'iterazione.

6.3 - Utilizzo degli iteratori

Esempi di utilizzo degli iteratori:

```
java
public class Comp {
    public static Poly diff(Poly p) throws NullPointerException {
        // EFFECTS: Se p è null lancia NullPointerException altrimenti
        //  ritorna il polinomio 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 zero
            q = q.add(new Poly(exp*p.coeff(exp), exp-1));
        }
        return q;
    }

    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) {
            Object p = g.next();
            if (p > m) return;
            System.out.println("The next prime is: " + p.toString());
        }
    }

    public static int max(Iterator g) throws EmptyException,
        NullPointerException {
        // REQUIRES: g contiene solo Integers
        // MODIFIES: g
        // EFFECTS: Se g è null lancia NullPointerException; se g è vuoto
        //  lancia EmptyException; altrimenti consuma tutti gli elementi
        //  di g e ritorna l'int più grande in g
        try {
            int m = ((Integer) g.next()).intValue();
```



```

        while (g.hasNext()) {
            int x = g.next();
            if (m < x) m = x;
        }
        return m;
    }
    catch (NoSuchElementException e) {
        throw new EmptyException("Comp.max");
    }
}
}

```

- Metodo `diff`: differenzia un `Poly`. Si noti che il codice non cattura la `NegativeExponentException` del costruttore `Poly` perché non chiama mai il costruttore con un esponente negativo (e l'eccezione non è controllata).
- Metodo `printPrimes`: utilizza l'iteratore `allPrimes`. Questa routine utilizza un ciclo di ricerca: l'iterazione si ferma non appena sono stati stampati abbastanza primi.
- Metodo `max` restituisce l'elemento più grande fornito dal suo generatore. Questa implementazione illustra altri modi di utilizzare i generatori. I generatori possono essere passati come argomenti alle routine. In questo caso, la routine **astrae dalla provenienza degli elementi**: potrebbero provenire da un insieme come `IntSet` o da un iteratore indipendente come `allPrimes`. Inoltre, il codice in `max` **innesca** il generatore: lo utilizza prima del ciclo per inizializzare l'iterazione.

Quindi, per ricapitolare:

- Il codice d'uso interagisce con un generatore tramite l'interfaccia `Iterator`
- Il codice che lo utilizza deve rispettare il vincolo imposto dalla clausola *requires* dell'iteratore
- I generatori possono essere passati come argomenti e restituiti come risultati
- A volte è utile **innescare** il generatore: consumare alcuni degli elementi prodotti prima di eseguire il loop sugli altri.

6.4 - Implementazione degli iteratori

Per implementare un iteratore, occorre scrivere il suo codice e definire e implementare una classe per il suo generatore. Ci sarà una classe generatore separata per ogni iteratore. Queste classi **non sono visibili agli utenti**: gli utenti non vedono le loro dichiarazioni di classe. Invece, il codice dell'utente viene scritto in termini del tipo generico `Iterator`.

Ogni nuova classe implementa l'interfaccia `Iterator`. Tali classi definiscono sottotipi di tipi definiti

dall'interfaccia `Iterator`. Pertanto, il codice scritto in termini di tipi di `Iterator` sarà in grado di utilizzare gli oggetti della classe. Per questo motivo non è necessario che gli utenti conoscano la nuova classe; è sufficiente conoscere l'interfaccia `Iterator` e la specifica dell'iteratore.

Ad esempio: l'implementazione dell'iteratore dei termini di Poly

```
java
public class Poly {
    private int[] trms;
    private int deg;

    public Iterator terms() { return new PolyGen(this); }

    // inner class
    private static class PolyGen implements Iterator {
        private Poly p; // il Poly da iterare
        private int n; // il prossimo termine da considerare

        PolyGen(Poly it) {
            // REQUIRES: it != null
            p = it;
            if (p.trms[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.trms[e] != 0) { n = e + 1; return new Integer(e); }
            throw new NoSuchElementException("Poly.terms")
        }

    } // fine PolyGen
}
```

L'oggetto generatore restituito è un oggetto di tipo `PolyGen`; la restituzione è legale perché `PolyGen` è un sottotipo di `Iterator`.

La classe `PolyGen` è implementata come una *classe statica interna*, cioè come una classe annidata all'interno di un'altra classe. Poiché `PolyGen` è privata, nessun codice esterno alla classe `Poly` sarà in grado di nominarla; pertanto, il codice che la utilizza non potrà dichiarare variabili di tipo `PolyGen` o costruire oggetti `PolyGen`. Invece, il codice utilizzatore otterrà gli oggetti `PolyGen` solo chiamando l'iteratore dei termini e li utilizzerà tramite l'interfaccia `Iterator`. Poiché `PolyGen` è una classe interna, il suo costruttore può essere richiamato dal

codice all'interno della classe `Poly` e il suo codice può accedere alle variabili di istanza private e ai metodi degli oggetti `Poly`. Questo è appropriato, poiché `PolyGen` è in realtà parte dell'implementazione di `Poly`, ossia la parte che fornisce l'iterazione di `terms`. Inoltre, la classe interna deve preservare l'invariante di rappresentazione di `Poly`, proprio come il codice `Poly`.

Nota bene: non viene fornita alcuna specifica per `PolyGen`, perché è già completamente specificato: i suoi oggetti devono essere generatori e devono obbedire alle specifiche dell'iteratore di termini.

Nota bene: l'eccezione lanciata nel metodo `next` identifica l'iteratore dei termini come la fonte del problema. Questo è appropriato perché gli utenti sono a conoscenza dell'iteratore, ma non della classe interna che implementa il generatore associato; pertanto, *l'informazione viene trasmessa* a un livello che ha senso per gli utenti.

Altro esempio: implementazione dell'iteratore `allPrimes`

```
java
public class Num {

    public static Iterator allPrimes() { return new PrimesGen(); }

    // classe interna
    private static class PrimesGen implements Iterator {
        private Vector ps; // numeri primi prodotti
        private int p; // prossimo candidato da provare

        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)
                int el = ((Integer) ps.get()).intValue();
                if (n%el == 0) break; // non ha alcun numero primo
                if (el*el > n) { // ha un numero primo
                    ps.add(new Integer(n));
                    p = n + 2;
                    return n;
                }
        }
    }
}
```

```
}  
}
```

Poiché tutti i primi sono dispari tranne il 2, il ciclo considera solo i numeri dispari come potenziali primi e il 2 viene trattato in modo particolare. L'implementazione conserva tutti i primi dispari generati finora nell'array `ps` e li utilizza per determinare se il prossimo candidato è un primo.

Nota bene: nell'intestazione del metodo `PrimesGen`, `next` non elenca `NoSuchElementException`, poiché non lancia questa eccezione. Tuttavia, le specifiche di `Iterator` indicano che questa eccezione può essere sollevata da `next`. È accettabile che il **metodo del sottotipo abbia meno eccezioni del metodo del supertipo** che risponde. Dal punto di vista dell'utente, questa regola ha senso: quando avviene la chiamata, l'utente è preparato a gestire le eccezioni elencate nell'intestazione del metodo che conosce. Se alcune di queste eccezioni non si verificano, non è un problema.

Quindi, per ricapitolare:

- L'implementazione di un iteratore richiede l'implementazione di una classe per il generatore associato.
- La classe generatore è una *classe statica interna*: annidata all'interno della classe che contiene l'iteratore e può accedere alle informazioni private della classe che la contiene.
- La classe generatore definisce un sottotipo di `Iterator`.
- L'implementazione del generatore presuppone che l'uso del codice rispetti i vincoli imposti dalla clausola *requires* dell'iteratore.

6.5 - Invarianti di rep e funzioni di astrazione per i generatori

È necessario definire invarianti e funzioni di astrazione per i generatori, proprio come si fa per i tipi astratti ordinari.

Gli **invarianti rep** per i generatori sono simili a quelli per i tipi astratti ordinari; l'unica differenza è che non forniremo un metodo per verificarli.

Ad esempio, l'invariante di rep per `PolyGen` è:

```
java  
// c.p ≠ 0 && (0 ≤ c.n ≤ c.p.deg)
```

Nota bene: questo invariante rep è espresso utilizzando variabili di istanza di `Poly`. Si noti anche come il requisito che `c.p` non sia nullo sia soddisfatto grazie alla clausola *requires* del costruttore di `PolyGen`.

Un altro esempio, è l'invariante rep di `PrimesGen`:

```
java
// c.ps non è null e
//  tutti gli elementi di c.ps sono primi, e sono ordinati in ordine
//  crescente, e includono tutti i numeri primi < c.p e > 2
```

Nota bene: questo invariante è piuttosto costoso da verificare!

Per definire la **funzione di astrazione** di un generatore, dobbiamo capire qual è lo stato astratto di un generatore. Tutti i generatori hanno lo **stesso stato astratto**: una sequenza di elementi che rimangono da generare. La funzione di astrazione deve quindi mappare il rappresentante in questa sequenza.

Ad esempio: funzioni di astrazione per `PrimesGen` e `PolyGen`:

```
java
// funzione di astrazione per PrimesGen
//  AF(c) = [p1, p2, ...] tale che
//  ogni pi è un intero e pi è un numero primo e pi >= c.p e
//  ogni numero primo >= c.p è nella sequenza e
//  pi > pj per tutti i > j >= 1.

// funzione di astrazione per PolyGen
//  AF(c) = [x1, ..., xn] tale che
//  ogni xi è un Integer e
//  ogni indice i >= n di un elemento non-zero di c.p.trms è nella
//  sequenza e nessun altro elemento è nella sequenza
//  e xi > xj per tutti i > j >= 1.
```

6.6 - Elenchi ordinati

Questa parte fornisce un altro esempio di iteratore. Questo iteratore fa parte di `OrderedIntList`, un'astrazione mutabile che mantiene i suoi elementi in ordine. L'iteratore `smallToBig` produrrà gli elementi della lista in questo ordine. Si noti che `OrderedIntList` non sarebbe adeguato senza l'iteratore, perché non ci sarebbe un modo conveniente per scoprire cosa c'è nell'elenco senza rimuovere elementi dall'elenco.

Le specifiche di `OrderedIntList` sono:

```
public class OrderedIntList {
    // OVERVIEW: Un elenco ordinato è un elenco ordinato mutabile di numeri //
    interi.
    // Un elenco tipico è una sequenza [x1, ..., xn] dove xi < xj se i < j.
```

```

// constructors
public OrderedIntList()
// EFFECTS: Inizializza this per essere una lista ordinata vuota.

// methods
public void addEl(int el) throws DuplicateException
// MODIFIES: this
// EFFECTS: Se el è in this, lancia DuplicateException;
// altrimenti, aggiunge el a this.

public void remEl(int el) throws NotFoundException
// MODIFIES: this
// EFFECTS: Se el non è in this, lancia NotFoundException;
// altrimenti, rimuove el da this.

public boolean isIn(int el)
// EFFECTS: Se el è in this ritorna true altrimenti ritorna false.

public boolean isEmpty()
// EFFECTS: Ritorna true se this è vuoto altrimenti ritorna false.

public int least() throws EmptyException
// EFFECTS: Se this è vuoto, lancia EmptyException;
// altrimenti, ritorna l'elemento più piccolo di this.

public Iterator smallToBig()
// EFFECTS: Ritorna un generatore che produrrà gli elementi di this
// (come Integers), esattamente una sola volta, dal più piccolo al più
// grande.
// REQUIRES: this non deve essere modificato mentre il generatore è in
// uso

public boolean repOk()
public String toString()
public Object clone()
}

```

Nota bene: I metodi `addEl` e `remEl` lanciano un'eccezione quando l'elemento è già presente nell'elenco ordinato. Questa scelta riflette la convinzione che gli utenti vorranno conoscere la situazione, senza doverla verificare esplicitamente (chiamando `isIn`). Inoltre, poiché è probabile che gli utenti effettuino chiamate che lanciano le eccezioni, le eccezioni devono essere controllate.

L'implementazione di `OrderedIntList` utilizza un **albero ordinato**. L'idea è che ogni nodo dell'albero contenga un valore e due sottonodi, uno a sinistra e uno a destra. I due sottonodi sono a loro volta elenchi ordinati e quindi la ripetizione è ricorsiva. L'albero è ordinato in modo che tutti i valori del sottonodo sinistro siano inferiori al valore del nodo padre e che tutti i valori del sottonodo destro siano maggiori del valore del nodo padre.

Una parte dell'implementazione è tale:

```
public class OrderedIntList {
    private boolean empty;
    private OrderedIntList left, right;
    private int val;

    public OrderedIntList() { empty = true; }

    public void addEl(int el) throws DuplicateException {
        if (empty) {
            left = new OrderedIntList( ); right = new OrderedIntList( );
            val = el; empty = false;
            return;
        }
        if (el == val) throw new DuplicateException("OrderedIntList.addEl");
        if (el < val) left.addEl(el); else right.addEl(el);
    }

    public void remEl(int el) throws NotFoundException {
        if (empty) throw new NotFoundException("OrderedIntList.remEl");
        if (el == val)
            try { val = right.least( ); right.remEl(val); }
            catch (EmptyException e) {
                empty = left.empty; val = left.val;
                right = left.right; left = left.left;
                return;
            }
        else if (el < val) left.remEl(el); else right.remEl(el);
    }

    public boolean isIn(int el) {
        if (empty) return false;
        if (el == val) return true;
        if (el < val) return left.isIn(el); else return right.isIn(el);
    }
}
```

```

    public boolean isEmpty() { return empty; }

    public int least() throws EmptyException {
        if (empty) throw new EmptyException("OrderedIntList.least");
        try { return left.least( ); }
        catch (EmptyException e) { return val; }
    }

    // ...

}

```

Nota bene: L'implementazione di `addEl` propaga implicitamente la `DuplicateException` sollevata dalle sue chiamate ricorsive; l'implementazione di `remEl` è simile.

L'iteratore `smallToBig` è implementato:

```

    public Iterator smallToBig() { return new OLGGen(this, count( )); }

    private int count ( ) {
        if (empty) return 0;
        return 1 + left.count( ) + right.count( );
    }

    // inner class
    private static class OLGGen implements Iterator {
        private int cnt; // conto del numero di elementi rimasti da generare
        private OLGGen child; // l'attuale sotto-generatore
        private OrderedIntList me; // il mio nodo

        OLGGen (OrderedIntList o, int n) {
            // REQUIRES: o != null
            cnt = n;
            if (cnt > 0) { me = o;
                child = new OLGGen(o.left, o.left.count( )); }
        }

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

        public Object next ( ) throws NoSuchElementException {
            if (cnt == 0)
                throw new
                    NoSuchElementException("OrderedIntList.smallToBig");
            cnt--;
        }
    }

```



```

    try { return new Integer(child.next( )); }
    catch (NoSuchElementException e) { }
    // se arriva qui, deve aver appena finito a sinistra;
    child = new OLGGen(me.right, cnt);
    return new Integer(me.val); }
} // fine di OLGGen

```

Il generatore inizia producendo gli elementi del sottoalbero sinistro. Quando tutti questi elementi sono stati prodotti, restituisce il valore del nodo superiore dell'albero e quindi produce gli elementi del sottoalbero destro. Poiché è importante che entrambi i metodi del generatore, e in particolare il metodo `hasNext`, vengano eseguiti in modo efficiente, l'implementazione tiene traccia di quanti elementi rimangono da produrre. Ciò avviene calcolando quanti elementi sono presenti nell'elenco al momento dell'inizio dell'iterazione.

La funzione di astrazione e l'invariante di rep per `OrderedIntList` sono:

```

// La funzione di astrazione è:
// AF(c) = se c.empty allora []
//          altrimenti AF(c.left) + [c.val] + AF(c.right)

// L'invariante di rep è:
// I(c) = c.empty || (c.left ≠ null && c.right ≠ null &&
//          I(c.left) && I(c.right) &&
//          (!c.left.isEmpty => c.left.greatest < c.val) &&
//          (!c.right.isEmpty => c.val < c.right.least) )

```

Qui `[]` è la sequenza vuota, `+` concatena le sequenze e `c.left.greatest` è l'elemento più grande di `c.left`.

Nota bene: sia la funzione di astrazione che l'invariante rep sono definiti in *modo ricorsivo*. È quello che ci si aspetta da un'implementazione ricorsiva!

La funzione di astrazione e l'invariante rep per il generatore sono:

```

// La funzione di astrazione è:
// AF(c) = se c.cnt = 0 allora []
//          altrimenti se |AF(c.child)| = c.cnt allora AF(c.child)
//          altrimenti AF(c.child) + [Integer(c.me.val)] + AF(OLGGen(c.right))

// L'invariante di rep è:
// I(c) = c.cnt = 0 || (c.cnt > 0 &&
//          c.me ≠ null && c.child ≠ null &&
//          (c.cnt = c.child.cnt + 1 ||
//          c.cnt = c.child.cnt + c.me.right.count + 1) )

```

Nota bene: l'invariante rep dipende dalla clausola *requires* dell'iteratore `smallToBig`, che richiede

che l'elenco ordinato non venga modificato mentre il generatore è in uso.

6.7 - Problemi di progettazione

La maggior parte dei tipi di dati include gli iteratori tra le proprie operazioni, soprattutto quelli come `IntSet` e `OrderedIntList`, i cui oggetti sono collezioni di altri oggetti. Gli iteratori sono spesso necessari per l'**adeguatezza**; rendono accessibili gli elementi di una collezione in modo efficiente e conveniente.

Un tipo può avere diversi iteratori.

Ad esempio, `OrderedIntList`, in aggiunta al metodo `smallToBig`, potrebbe avere il metodo `bigToSmall`:

```
java
public Iterator bigToSmall()
    // EFFECTS: Ritorna un generatore che produrrà gli elementi di this
    // (come Integers), esattamente una sola volta, dal più grande al
    // più piccolo.
    // REQUIRES: this non deve essere modificato mentre il generatore
    // è in uso
```

Per le collezioni mutabili, abbiamo sempre richiesto che il corpo del ciclo non modifichi la collezione su cui si itera. Se si omette questo requisito, il generatore restituito dall'iteratore deve comportarsi in un modo ben definito anche quando si verificano modifiche.

Ad esempio, supponiamo che l'intero `n` venga cancellato da un `IntSet` mentre il generatore restituito da `elements` è in uso; `n` deve essere prodotto dal generatore o no?

Un approccio è quello di richiedere che un generatore produca gli elementi contenuti nel suo argomento di raccolta nel momento in cui viene creato dall'iteratore, anche se le modifiche avvengono successivamente. Il comportamento di un generatore specificato in questo modo è ben definito, ma l'implementazione rischia di essere inefficiente.

Ad esempio, se l'iteratore `elements` dovesse restituire questo tipo di generatore, la sua implementazione dovrebbe fornire al generatore una copia dell'array `els`, proprio quello che abbiamo contestato nel metodo `members`. Poiché l'approccio di vincolare il corpo del ciclo evita tali inefficienze, sarà preferito nella maggior parte dei casi.

Una questione correlata è se l'iteratore o il generatore che restituisce possono modificare l'insieme. Come convenzione generale, **tali modifiche dovrebbero essere evitate**.

Le modifiche del corpo del ciclo o dell'iteratore o del generatore possono talvolta essere utili.

Ad esempio, si consideri un programma che esegue compiti in attesa su una coda di compiti:

```
java
Iterator g = q.allTasks();
while (g.hasNext()) {
    Task t = (Task) g.next();
    // eseguire t
    // se t genera un nuovo compito nt,
    // lo mette in coda eseguendo q.enq(nt)
}
```

Quando l'attività in corso genera un'altra attività, è sufficiente metterla in coda per eseguirla successivamente; il generatore restituito dall'iteratore `allTasks` la presenterà per l'esecuzione al momento opportuno. Tuttavia, esempi come questo sono rari; di solito né il generatore né il corpo del ciclo modificano l'insieme.

7 - Gerarchia dei tipi

Tutti i membri di una famiglia hanno un comportamento simile: tutti hanno determinati metodi e le chiamate a tali metodi si comportano in modo simile. I membri della famiglia possono differenziarsi estendendo il comportamento dei metodi comuni o fornendo metodi aggiuntivi.

Una famiglia di tipi può corrispondere al tipo di gerarchia che si trova nel mondo reale.

Per esempio, gli `autobus` e le `automobili` sono entrambi tipi specializzati di `veicoli`, o i `cani` e i `gatti` sono tipi speciali di `mammiferi`.

Oppure può corrispondere a concetti che esistono solo all'interno dei programmi.

Per esempio, un `BufferedReader` è un tipo specializzato di `Reader`.

Una famiglia di tipi è definita da una **gerarchia di tipo**.

La **gerarchia dei tipi** viene utilizzata per definire famiglie di tipi. In cima alla gerarchia c'è un tipo la cui specifica definisce il comportamento comune a tutti i membri della famiglia, incluse le firme e il comportamento di tutti i metodi comuni. Gli altri membri della famiglia sono definiti come **sottotipi** di questo tipo, che viene indicato come il loro **supertipo**. La gerarchia può essere di **più di due livelli**: i sottotipi possono avere a loro volta dei sottotipi, e così via.

Le famiglie di tipi vengono utilizzate in due modi diversi.

Possono essere usate per definire **più implementazioni** di un tipo. In questo caso, i sottotipi non aggiungono alcun nuovo comportamento, a parte il fatto che ognuno di essi ha i propri costruttori. Piuttosto, la classe che implementa il sottotipo implementa esattamente il comportamento definito dal supertipo.

Ad esempio, si potrebbe usare una famiglia di tipi per fornire polinomi sia radi che densi, in modo da utilizzare la rappresentazione più efficiente per ogni oggetto polinomiale.

Più in generale, però, i sottotipi di una famiglia di tipi **estendono il comportamento** dei loro supertipi, ad esempio fornendo metodi aggiuntivi. La gerarchia che definisce una famiglia di tipi può essere a più livelli. Inoltre, alla base della gerarchia possono esserci più implementazioni di un sottotipo.

La gerarchia dei tipi richiede che i membri della famiglia di tipi abbiano comportamenti correlati. In particolare, il comportamento del supertipo deve essere supportato dai sottotipi: gli oggetti del sottotipo possono essere **sostituiti** da quelli del supertipo senza influenzare il comportamento del codice che li utilizza. Questa proprietà è chiamata **principio di sostituzione**. Consente di scrivere il codice d'uso in termini di specifiche del supertipo, pur funzionando correttamente quando si utilizzano oggetti del sottotipo.

Ad esempio, il codice può essere scritto in termini di tipo `Reader`, ma funziona correttamente

quando si utilizza un `BufferedReader`.

Il principio di sostituzione fornisce **l'astrazione tramite la specificazione** di un tipo famiglia. Ci permette di **astrarre dalle differenze** tra i sottotipi per concentrarci sui punti in comune, che vengono catturati nella specifica del supertipo.

7.1 - Assegnamento e Dispatching

L'utilità della gerarchia dei tipi si basa su un allentamento delle regole che governano l'assegnazione, il passaggio degli argomenti e sul modo in cui le chiamate vengono distribuite al codice.

7.1.1 - Assegnazione

Una variabile dichiarata come appartenente a un tipo può in realtà riferirsi a un oggetto appartenente a un sottotipo di quel tipo. In particolare, se S è un sottotipo di T , gli oggetti S possono essere assegnati a variabili di tipo T e possono essere passati come argomenti o risultati dove ci si aspetta una T .

Per esempio, supponiamo che `DensePoly` e `SparsePoly` siano sottotipi di `Poly`, dove:

- `DensePoly` fornisce una buona implementazione delle `Poly` che hanno un numero relativamente basso di coefficienti nulli sotto il termine di grado,
- `SparsePoly` è buona per le `Poly` che non soddisfano il criterio di `DensePoly`

Quindi è consentito il seguente codice:

```
java
Poly p1 = new DensePoly(); // il Poly zero
Poly p2 = new SarsePoly(3, 20); // il Poly 3x^{20}
```

Pertanto, le variabili di tipo `Poly` possono riferirsi a oggetti `DensePoly` e `SparsePoly`

Assegnazioni di questo tipo significano che il tipo di oggetto a cui fa riferimento una variabile non è necessariamente lo stesso di quello dichiarato per la variabile.

Ad esempio, `p1` è dichiarato di tipo `Poly`, ma in realtà si riferisce a un oggetto `DensePoly`.

Per distinguere questi due tipi, si parla di tipo *apparente* (*apparent type*) di un oggetto e di tipo *effettivo* (*actual type*).

- Il **tipo apparente** è quello che il compilatore può dedurre dalle informazioni disponibili (dalle dichiarazioni);
- Il **tipo effettivo** è quello che l'oggetto ha realmente.

Ad esempio, l'oggetto a cui fa riferimento `p1` ha tipo apparente `Poly` ma tipo effettivo `DensePoly`.

Ricorda: Il tipo effettivo di un oggetto sarà sempre un sottotipo del suo tipo apparente

Il compilatore esegue il **controllo** dei tipi sulla base delle informazioni disponibili: utilizza i tipi apparenti, non i tipi effettivi, per eseguire il controllo. In particolare, determina quali chiamate di metodo sono legali in base al tipo apparente:

Ad esempio:

```
java
int d = p1.degree();
```

è considerato legale poiché `Poly`, il tipo apparente di `p1`, ha un metodo chiamato `degree` che non richiede argomenti e restituisce un `int`.

L'obiettivo del controllo è garantire che, quando viene eseguita una chiamata di metodo, l'oggetto abbia effettivamente **un metodo con la firma appropriata**. Affinché ciò abbia senso, è essenziale che l'oggetto a cui si riferisce `p1` abbia tutti i metodi indicati dal supertipo con le firme previste.

Pertanto, `DensePoly` e `SparsePoly` devono avere tutti i metodi dichiarati per `Poly` con le firme previste. Nell'esempio precedente, supponiamo che `Poly` non abbia il metodo `degree`. In questo caso, la chiamata sarà rifiutata dal compilatore anche se l'oggetto a cui fa riferimento `p1` ha effettivamente tale metodo.

Un oggetto appartenente a un sottotipo viene creato nel codice che sa di avere a che fare con il sottotipo. Tale codice può utilizzare i metodi extra del sottotipo. Ma il codice scritto in termini di supertipo può usare solo i metodi del supertipo.

7.1.2 - Dispatching

Il compilatore potrebbe non essere in grado di determinare quale codice eseguire quando viene chiamato un metodo. Il codice da eseguire dipende dal tipo effettivo dell'oggetto, mentre il compilatore conosce solo il tipo apparente.

Ad esempio:

```
java
static Poly diff(Poly p) {
    // differenzia p
    Iterator g = p.terms();
}
```

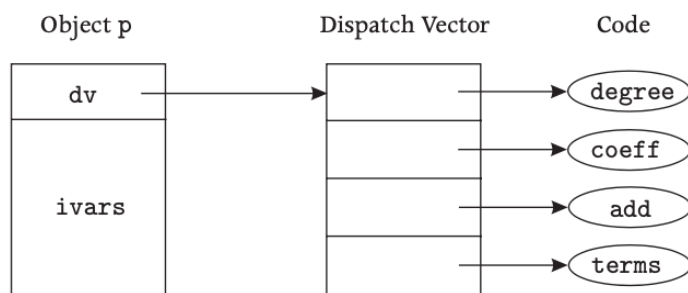
Quando questa routine viene compilata, il compilatore non sa se il tipo effettivo dell'oggetto a cui si riferisce `p` è un `DensePoly` o uno `SparsePoly`, tuttavia deve chiamare l'implementazione di `terms` per `DensePoly` se `p` è un `DensePoly` e l'implementazione di `terms` per `SparsePoly` se `p` è uno `SparsePoly`. In altre parole, deve chiamare il codice determinato dal tipo effettivo, poiché le rappresentazioni sono diverse e il codice funziona in modo diverso nei due casi.

Come già discusso precedentemente, la chiamata del metodo giusto si ottiene con un meccanismo di runtime chiamato **dispatching**. Il dispatching fa sì che le **chiamate ai metodi siano indirizzate al codice effettivo** dell'oggetto, cioè al codice fornito dalla sua classe.

Il compilatore non genera il codice per chiamare direttamente il metodo. Al contrario, **genera il codice** per trovare il codice del metodo e poi **creare una diramazione** verso di esso.

Esistono diversi modi per implementare il dispatching. Un approccio consiste nel far sì che gli oggetti contengano, oltre alle loro variabili di istanza, un **puntatore a un vettore dispatch**, che contiene i puntatori alle implementazioni dei metodi dell'oggetto.

Un esempio di tale approccio utilizzando l'oggetto Poly :



Il codice chiamante di un metodo recupera il *vettore dispatch* dall'oggetto, recupera l'indirizzo del codice del metodo dallo slot appropriato del *vettore dispatch* e si dirama a quell'indirizzo.

Ad esempio, per chiamare il metodo `terms` di `Poly`, `p`, il codice chiamante chiamerà il codice puntato dal quarto slot del *vettore di dispatch* di `p`.

7.2 - Definizione di una gerarchia di tipi

Il primo passo nella definizione di una gerarchia di tipi è definire il tipo al **vertice della gerarchia**.

Questo tipo ha una specifica simile a quelle che conosciamo, tranne che per il fatto che può essere incompleta, ad esempio priva di costruttori.

Le specifiche dei sottotipi sono fornite rispetto a quelle dei loro supertipi. Piuttosto che ripetere le parti della specifica del supertipo che non cambiano, **le specifiche dei sottotipi si concentrano sulle novità**.

Pertanto, la specifica del sottotipo deve definire i costruttori del sottotipo, più eventuali metodi extra forniti dal sottotipo. Inoltre, se il sottotipo modifica il comportamento di alcuni metodi del supertipo, è necessario fornire le specifiche per tali metodi. Sono consentite solo **modifiche limitate al comportamento dei metodi dei supertipi**.

Le implementazioni dei supertipi sono solitamente diverse da quelle viste finora. **Alcuni supertipi non sono affatto implementati**; altri possono avere solo implementazioni parziali, in cui alcuni metodi sono implementati ma altri no. Inoltre, l'implementazione di un supertipo può fornire informazioni extra a potenziali sottotipi, dando loro accesso a variabili di istanza o a metodi che gli utenti non possono chiamare.

Quando i supertipi sono implementati, anche solo parzialmente, i sottotipi sono implementati come estensioni dell'implementazione del supertipo. Le repliche degli oggetti sottotipo contengono al loro interno le variabili di istanza definite nell'implementazione del supertipo. Alcuni metodi dei sottotipi sono ereditati dall'implementazione del supertipo e non devono essere implementati dall'implementazione del sottotipo. Tuttavia, l'implementazione del sottotipo può anche reimplementare questi metodi.

7.3 - Definire le gerarchie in Java

Le gerarchie di tipi vengono definite in Java utilizzando il **meccanismo dell'ereditarietà**. Questo meccanismo consente a una classe di essere una *sottoclasse* di un'altra classe (la sua *superclasse*) e di implementare zero o più interfacce. I supertipi in Java sono definiti sia dalle classi che dalle interfacce. In entrambi i casi, la classe o l'interfaccia forniscono una specifica per il tipo.

Un'**interfaccia definisce solo la specifica**; non contiene alcun codice per l'implementazione del supertipo. Quando un supertipo è definito da una **classe**, oltre alla specifica, la classe può fornire un'**implementazione completa o parziale**.

In Java esistono due tipi di classi: **le classi concrete** e **le classi astratte**. Le classi concrete forniscono un'implementazione completa del tipo. Le classi astratte forniscono al massimo un'implementazione parziale del tipo. *Non hanno* oggetti (poiché alcuni dei loro metodi non sono ancora implementati) e il codice non può chiamare i loro costruttori.

Entrambi i tipi di classi possono contenere metodi normali (come quelli visti finora) e *metodi finali*. I metodi finali non possono essere reimplementati dalle sottoclassi (non vengono utilizzati in questo libro);

Le classi astratte possono inoltre avere **metodi astratti**; si tratta di metodi che non sono implementati dalla superclasse e, pertanto, devono essere implementati da qualche sottoclasse. Tuttavia, la distinzione tra queste categorie di metodi è di interesse *solo* per gli implementatori delle sottoclassi; **non è di interesse per gli utenti**.

Una sottoclasse dichiara la sua superclasse affermando nella sua intestazione che **extends quella classe**. In questo modo, avrà automaticamente tutti i metodi della sua superclasse con gli stessi nomi e firme definiti nella superclasse. Inoltre, può fornire alcuni *metodi extra*.

Una sottoclasse concreta deve contenere le implementazioni dei costruttori della sottoclasse e dei metodi extra. Inoltre, deve implementare i metodi astratti della sua superclasse e può reimplementare o *sovrascrivere* i metodi normali. *Ingerisce* dalla sua superclasse le implementazioni dei metodi finali e di tutti i metodi normali che non sovrascrive. I metodi che sovrascrive devono avere firme identiche a quelle definite dalla superclasse, tranne per il fatto che il metodo della sottoclasse può lanciare un numero minore di tipi di eccezione.

La **rappresentazione di un oggetto** della sottoclasse consiste nelle variabili di istanza dichiarate per

la superclasse e in quelle dichiarate per la sottoclasse. Quando si implementa la sottoclasse, può essere necessario avere accesso ai dettagli di rappresentazione dell'implementazione della superclasse. Ciò sarà possibile solo se la superclasse renderà accessibili alla sottoclasse parti della sua implementazione. Un aspetto importante nella progettazione di una superclasse è determinare l'interfaccia che essa fornisce alle sue sottoclassi. La cosa migliore è che le sottoclassi possano interagire con le superclassi interamente tramite l'**interfaccia pubblica**, in quanto ciò preserva l'astrazione completa e consente alla superclasse di essere reimplementata senza influire sulle implementazioni delle sue sottoclassi. Ma questa interfaccia può essere inadeguata a consentire sottoclassi efficienti. In questo caso, la superclasse può dichiarare metodi, costruttori e variabili di istanza **protetti**, visibili alle sottoclassi. Tuttavia, i membri protetti sono anche visibili al pacchetto: sono accessibili ad altre parti del pacchetto della superclasse. Pertanto, la loro visibilità non è così limitata come si vorrebbe: il rappresentante è esposto ad altro codice del pacchetto.

Ricorda: Ogni classe che non estende esplicitamente un'altra classe estende implicitamente `Object`. Quindi, ogni classe è una sottoclasse di `Object` e deve fornire implementazioni corrette per i metodi di `Object`.

7.4 - Un semplice esempio

Il primo esempio riguarda una famiglia di tipi di insiemi di interi. In questo caso, la classe in cima alla gerarchia non è astratta: `IntSet` fornisce un insieme minimo (adeguato) di metodi; i sottotipi forniranno metodi aggiuntivi. `IntSet` ha solo metodi normali che implementa e che le sue sottoclassi possono sovrascrivere:

```
public class IntSet {
    // OVERVIEW: Gli IntSet sono insiemi mutabili e non limitati di numeri
    // interi.
    // Un IntSet tipico è {x1, . . . , xn}.

    // constructors
    public IntSet()
    // EFFECTS: Inizializza this per essere vuoto.

    // methods
    public void insert(int x)
    // MODIFIES: this
    // EFFECTS: Aggiunge x agli elementi di of this.

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

```

public boolean isIn(int x)
// EFFECTS: Se x è in this ritorna true altrimenti ritorna false.

public int size()
// EFFECTS: Returns la cardinalità di this.

public Iterator elements()
// EFFECTS: Ritorna un generatore che produce tutti gli elementi di
// this (come Integers), esattamente una sola volta, in ordine
// arbitrario.
// REQUIRES: this non deve essere modificato mentre il generatore
// è in uso.

public boolean subset(IntSet s)
// EFFECTS: Ritorna true se this è un sottoinsieme di s altrimenti
// ritorna false.

public boolean repOk()
}

```

Una parte di implementazione può essere così:

```

public class IntSet {
    private Vector els; // the elements

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

    private int getIndex(Integer x) { ... }

    public boolean isIn(int x) {
        return getIndex(new Integer(x)) >= 0;
    }

    public boolean subset(IntSet s) {
        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;
    }

    // l'implementazione di altri metodi vanno qua
}

```

Gli elementi di `IntSet` sono memorizzati nel vettore `els`. Il punto principale da notare è che non ci sono membri protetti. Ciò significa che le sottoclassi di `IntSet` non hanno accesso speciale ai componenti della parte della superclasse della loro *rep*. Questa mancanza di accesso è accettabile in questo caso, perché l'iteratore `elements` fornisce una potenza adeguata.

Consideriamo ora un tipo `MaxIntSet`, che è un sottotipo di `IntSet`.

```
public class MaxIntSet extends IntSet {
    // OVERVIEW: MaxIntSet è un sottotipo di IntSet con un metodo
    // aggiuntivo, max, per determinare l'elemento massimo dell'insieme.

    // constructors
    public MaxIntSet()
    // EFFECTS: Rende this un MaxIntSet vuoto.

    // methods
    public int max() throws EmptyException
    // EFFECTS: Se this è vuoto lancia EmptyException altrimenti ritorna
    // l'elemento più grande di this.
}
```

Un oggetto `MaxIntSet` si comporta come un oggetto `IntSet`, ma ha un metodo in più che restituisce l'elemento più grande dell'insieme. Si noti che la specifica si basa su quella di `IntSet` e definisce solo ciò che è nuovo, sia nella clausola generale che nelle specifiche delle operazioni. Nel caso di `MaxIntSet`, non sono state modificate le specifiche dei metodi del supertipo e quindi sono state specificate solo le nuove operazioni, il metodo `max` e il costruttore `MaxIntSet`. Tutti gli altri metodi hanno le specifiche previste dalle specifiche di `IntSet`.

Un modo semplice per implementare `MaxIntSet` è permettere a `IntSet` di tenere traccia degli elementi dell'insieme. Tuttavia, per facilitare la ricerca dell'elemento massimo, è auspicabile che la sottoclasse abbia una variabile di istanza:

```
private int biggest; // l'elemento massimo (se l'insieme non è vuoto)
```

Pertanto, gli oggetti `MaxIntSet` hanno due variabili di istanza: `els` (dalla superclasse) e `biggest` (dalla sottoclasse).

Quando si inserisce un nuovo elemento, se questo è più grande dell'attuale `biggest`, il valore di `biggest` viene modificato. Quando un elemento viene rimosso, se è il più grande, è necessario reimpostare `biggest` per mantenere il nuovo massimo. Il calcolo del nuovo `biggest` può essere effettuato utilizzando l'iteratore `elements`:

```

public class MaxIntSet extends IntSet {
    private int biggest; // l'elemento più grande se set non è vuoto

    public MaxIntSet() { super(); }

    public void insert(int x) {
        if(size() == 0 || x > biggest) biggest = x;
        super.insert(x);
    }

    public void remove(int x) {
        super.remove(x);
        if (size() == 0 || x < biggest) return;
        Iterator g = elements();
        biggest = ((Integer) g.next()).intValue();
        while (g.hasNext()) {
            int z = ((Integer) g.next()).intValue();
            if (z > biggest) biggest = z;
        }
    }

    public int max() throws EmptyException {
        if (size() == 0) throw new EmptyException ("MaxIntSet.max");
        return biggest;
    }

    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 > biggest) return false;
            if (z == biggest) found = true;
        }
        return found;
    }
}

```

La classe implementa il costruttore e il metodo `max`. Inoltre, sovrascrive le implementazioni di `insert`, `remove` e `repOk`. Tuttavia, le implementazioni di `size`, `isIn`, `elements`, `subset` e `toString` sono ereditate da `IntSet`.

Innanzitutto, si noti l'implementazione del costruttore. La prima cosa che un costruttore di sottoclasse deve fare è **chiamare un costruttore di superclasse** per inizializzare le variabili di istanza della superclasse; in questo caso la chiamata viene fatta esplicitamente (usando la sintassi `super()`). Se il costruttore della sottoclasse non contiene questa chiamata, Java inserirà automaticamente una chiamata al costruttore della superclasse senza argomenti. Pertanto, in questo caso, la chiamata potrebbe essere omessa, ad esempio:

```
public MaxIntSet() {}
```

è anche un'implementazione corretta del costruttore. Tuttavia, se è necessaria una chiamata a un costruttore di una superclasse che ha argomenti, la chiamata deve essere fatta esplicitamente. In ogni caso, quando il resto del costruttore di `MaxIntSet` inizia a funzionare, l'array `els` è già stato inizializzato. A questo punto, il costruttore di `MaxIntSet` non deve svolgere alcun lavoro, poiché `biggest` non ha alcun valore quando l'insieme è vuoto.

Per implementare i metodi di `MaxIntSet`, dobbiamo utilizzare i **metodi sovrascritti** di `IntSet`. Ad esempio, per `insert`, vogliamo che sia il metodo `insert` di `IntSet` a svolgere il lavoro effettivo. All'interno di una sottoclasse, tutti i metodi sovrascritti della superclasse sono disponibili. Tuttavia, c'è un problema di denominazione: come distinguere il metodo sovrascritto (quello che viene implementato per `MaxIntSet`) dal metodo sovrascritto? Java risolve questo problema utilizzando una **forma composta**: ad esempio, `super.insert`, per denominare il metodo sovrascritto. Un nome senza prefisso, ad esempio `insert`, indica il metodo sovrascritto, così come la forma `this.insert`. Esempi di utilizzo di metodi sovrascritti si hanno nelle implementazioni di `insert` e `remove`. Sebbene i metodi sovrascritti siano visibili alle sottoclassi, *non sono* accessibili agli utenti degli oggetti della sottoclasse. Ad esempio, se `x` è un oggetto `MaxIntSet`, `x.insert` nomina l'implementazione di `insert` fornita da `MaxIntSet` e non c'è modo di usare il codice per nominare il metodo `insert` fornito da `IntSet`.

L'**invariante rep** e la **funzione di astrazione** di una sottoclasse sono tipicamente definiti in termini di quelli della superclasse. Si ha quindi

```
// La funzione di astrazione è:  
// AF_MaxIntSet(c) = AF_IntSet(c)
```

Qui abbiamo introdotto una notazione per distinguere le due funzioni di astrazione: quella per `IntSet` è `AF_IntSet` e quella per `MaxIntSet` è `AF_MaxIntSet`. In questo esempio, il campo più grande non influisce sull'insieme che un oggetto `MaxIntSet` rappresenta; pertanto, `AF_MaxIntSet` produce semplicemente lo stesso insieme di `AF_IntSet`.

Nota bene: è ragionevole applicare `AF_IntSet` a un oggetto `MaxIntSet`, poiché tale oggetto possiede tutte le variabili di istanza di `IntSet`.

Il fatto che le due funzioni di astrazione siano uguali riflette il fatto che `MaxIntSet` si basa su `IntSet` per memorizzare gli elementi dell'insieme.

L'invariante di rappresentazione per `MaxIntSet` è:

```
// L'invariante di rappresentazione è:  
// I_MaxIntSet(c) = c.size > 0 =>  
//      (c.biggest in AF_IntSet(c) &&  
//      per tutti gli x in AF_IntSet(c) (c <= c.biggest))
```

Quindi, l'invariante di rappresentazione è definito in termini di funzione di astrazione per `IntSet`.

Nota bene: non include l'invariante `rep` di `IntSet` per il semplice motivo che preservare tale invariante è compito dell'implementazione di `IntSet` e non c'è modo che l'implementazione di `MaxIntSet` possa interferire, poiché ha solo accesso pubblico alla parte `IntSet` della sua `rep`. D'altra parte, l'implementazione di `repOk` per una sottoclasse dovrebbe sempre controllare l'invariante della superclasse, poiché la `rep` della sottoclasse non può essere corretta se la parte della `rep` della superclasse non è corretta.

L'implementazione di `remove` potrebbe non essere soddisfacente, poiché a volte deve passare due volte attraverso l'array `els`: una volta per rimuovere `x` e un'altra per ricompilare `biggest`. Per fare meglio, tuttavia, `MaxIntSet` richiederebbe l'accesso alla rappresentazione di `IntSet`, che potrebbe essere realizzato facendo in modo che `IntSet` dichiari `els` come `protected`. In questo caso, l'invariante `rep` di `MaxIntSet` deve includere l'invariante `rep` di `IntSet` (poiché l'implementazione di `MaxIntSet` potrebbe causare la violazione dell'invariante `rep`), dando:

```
// L'invariante di rappresentazione è:  
// I_MaxIntSet(c) = I_IntSet(c) && c.size > 0 =>  
//      (c.biggest in AF_IntSet(c) &&  
//      per tutti gli x in AF_IntSet(c) (x <= c.biggest) )
```

Quindi, per ricapitolare:

- La **funzione di astrazione** di una sottoclasse `AF_sub` è tipicamente definita utilizzando `AF_super`, ovvero la funzione di astrazione della superclasse
- L'**invariante di rappresentazione** della sottoclasse, `I_sub`, deve includere un controllo sull'invariante `rep` della superclasse, `I_super`, solo se la superclasse ha alcuni membri protetti. Tuttavia, `repOk` per la sottoclasse dovrebbe sempre controllare `repOk` per la superclasse.

7.5 - Tipi di eccezione

I tipi di eccezione sono sottotipi di `Throwable` e l'implementazione di `Throwable` fornisce metodi che accedono alla stringa dell'oggetto eccezione. È quindi possibile implementare nuovi tipi di eccezione semplicemente definendo i loro costruttori. È anche possibile definire un tipo di eccezione

che abbia metodi aggiuntivi e che contenga più informazioni nei suoi oggetti.

Ad esempio:

```
java
public MyException extends Exception {
    // OVERVIEW: Gli oggetti MyException contengono un int e una stringa.
    private int val;

    public MyException(String s, int v) { super(s); val = v; }
    public MyException(int v) { super(); val = v; }
    public int valueOf() { return val; }
}
```

I suoi costruttori richiedono un argomento `int` e il suo metodo, `valueOf`, consente ai programmi di accedere all'`int`.

7.6 - Classi astratte

Una **classe astratta** fornisce solo un'**implementazione parziale** di un tipo. Può avere alcune variabili di istanza e, se le ha, avrà anche uno o più costruttori. Questi costruttori non possono essere chiamati dai suoi utenti, poiché una classe astratta non ha oggetti, ma i costruttori possono essere usati dalle sottoclassi per inizializzare la parte di rep. della superclasse.

In genere, una classe astratta contiene sia **metodi astratti** che **metodi normali** (non astratti). Fornisce le implementazioni dei metodi non astratti. Queste implementazioni spesso fanno uso dei metodi astratti, il che consente alla superclasse di definire la parte generica dell'implementazione, mentre le sottoclassi si occupano dei dettagli. Ciò viene definito come **template pattern**.

Implementare i metodi nella superclasse è auspicabile: li implementiamo una sola volta, anche se ci possono essere molte sottoclassi. Le sottoclassi non solo avranno meno codice, ma saranno anche più facili da correggere.

Per esempio, supponiamo di voler definire un tipo `SortedIntSet`, che è come un `IntSet`, tranne per il fatto che l'iteratore degli elementi fornisce accesso agli elementi in ordine:

```
java
public class SortedIntSet extends IntSet {
    // OVERVIEW: Un insieme int ordinato è un insieme int i cui elementi
    // sono accessibili in ordine.

    // constructors:
    public SortedIntSet( )
        // EFFECTS: Rende this un insieme ordinato vuoto

    // methods:
```

```

public Iterator elements()
    // EFFECTS: Restituisce un generatore che produce tutti gli
    //          elementi di questo, ciascuno esattamente una volta, in
    //          ordine crescente.
    // REQUIRES: this non è modificabile mentre il generatore è in uso

public int max() throw EmptyException
    // EFFECTS: Se this è vuoto lancia EmptyException altrimenti
    //          ritorna l'elemento più grande di this.

public boolean subset(SortedIntSet s)
}

```

Nota bene: viene fornita una specifica per `elements`, poiché la sua specifica è cambiata: produce gli elementi in maniera ordinata. Si noti anche che `SortedIntSet` fornisce un metodo di sottoinsieme aggiuntivo e quindi il suo metodo di sottoinsieme è sovraccaricato. Ha due metodi di sottoinsieme:

```

java
public boolean subset(IntSet s) // ingerito
public boolean subset(SortedIntSet s) // extra

```

Poiché non viene fornita alcuna specifica per il metodo `subset` aggiuntivo, esso deve avere le stesse specifiche del metodo `subset` ereditato. Il motivo per cui viene fornito un secondo metodo `subset` è quello di ottenere prestazioni migliori nel caso in cui si sappia che l'argomento è un `SortedIntSet`.

Per implementare `SortedIntSet`, si potrebbe utilizzare un elenco ordinato. Tuttavia, se `SortedIntSet` è implementato da una sottoclasse di `IntSet` abbiamo un problema: ogni oggetto `SortedIntSet` conterrà al suo interno variabili di istanza ereditate da `IntSet`. Queste variabili di istanza non sono più interessanti, poiché non vogliamo mantenere gli elementi di un `SortedIntSet` nel vettore `els`.

È possibile ottenere sottotipi efficienti i cui oggetti non contengano variabili di istanza inutilizzate, non avendo queste variabili nella superclasse.

Tuttavia, se la classe `IntSet` non ha un modo per memorizzare gli elementi dell'insieme, non può avere alcun oggetto. Pertanto, deve essere astratta:

```

java
public abstract class IntSet {
    protected int sz; // la grandezza

    // constructors
    public IntSet() { sz = 0; }
}

```



```

// abstract methods
public abstract void insert(int x);
public abstract void remove(int x);
public abstract Iterator elements();
public abstract boolean repOk();

// methods
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; }

// le implementazioni di "subset" e "toString" vanno qui
}

```

Qui `insert`, `remove`, `elements` e `repOk` sono astratti. `isIn`, `subset` e `toString` sono implementati utilizzando uno dei metodi astratti (`elements`). Anche se la dimensione potrebbe essere implementata utilizzando `elements`, ciò sarebbe poco efficiente.

Inoltre, tutte le sottoclassi avranno bisogno di un modo per implementare la dimensione in modo efficiente. Pertanto, `IntSet` dispone di una variabile di istanza, `sz`, che tiene traccia della dimensione. Ciò significa che il definitore di `IntSet` deve decidere se renderla accessibile o di nascondere, fornendo l'accesso attraverso metodi protetti. Se lo nascondiamo, `IntSet` può mantenere l'invariante:

```

java
sz >= 0

```

Tuttavia, questo non è interessante: ciò che conta è che `sz` è la dimensione dell'insieme e questo può essere mantenuto solo dalle sottoclassi. Pertanto, consentiremo alle sottoclassi di accedere direttamente a `sz`; per questo motivo è dichiarato *protetto*. Poiché `IntSet` non garantisce l'invariante `rep`, il suo metodo `repOk` è **astratto**.

Nota bene: la classe non ha alcuna funzione di astrazione; questo è tipico di una classe astratta, poiché le implementazioni reali sono fornite dalle sottoclassi.

Un'implementazione parziale di `SortedIntSet` come sottoclasse di `IntSet`:

```
java
public class SortedIntSet extends IntSet {
    private OrderedIntList els;
    // La funzione di astrazione è:
    //  AF(c) = c.els[1], ..., c.els[c.sz]
    // L'invariante di rappresentazione è:
    //  c.els != null && c.sz = c.els.size

    public SortedIntSet() { els = new OrderedIntList(); }

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

    public Iterator elements() { return super.subset(s); }

    public boolean subset(SortedIntSet s) {
        // l'implementazione qui presente sfrutta il fatto che smallToBig
        // di OrderedIntList restituisce gli els in ordine crescente
        ...
    }

    // le implementazioni di insert, remove e repOk vanno qua
}
```

Questa sottoclasse deve implementare tutti i metodi astratti, ma può ereditare i metodi non astratti come `size`. La sottoclasse utilizza il tipo `OrderedIntList`.

Nota bene: l'implementazione del metodo `subset` extra può essere più efficiente di quella del metodo `subset` ereditato; il metodo extra sarà chiamato quando l'oggetto e l'argomento hanno entrambi il tipo apparente `SortedIntSet`. Si noti inoltre che il metodo ereditato viene sovrascritto in modo da poter fornire un'implementazione più efficiente quando l'argomento è un `SortedIntSet`.

La funzione di astrazione mappa l'elenco ordinato `els` in un insieme; tratta l'elenco ordinato come una sequenza, come descritto nelle specifiche di `OrderedIntList`, e utilizza la notazione `[]` per accedere agli elementi della sequenza. L'invariante `rep` vincola sia la variabile di istanza `SortedIntSet`, `els` e la variabile di istanza `sz` presente in `IntSet`.

Nota bene: si assume che `els` sia ordinato, poiché ciò è vero per tutti gli oggetti `OrderedIntList`.

Anche le sottoclassi possono essere astratte. Possono continuare a elencare alcuni dei metodi astratti della superclasse come astratti, oppure possono introdurne di nuovi.

Quindi, per ricapitolare:

- È auspicabile **evitare l'uso di membri protetti** per due motivi:
 1. Senza di essi, la superclasse può essere reimplementata senza influenzare l'implementazione di qualsiasi sottoclasse;
 2. I membri protetti sono visibili nel pacchetto, il che significa che altro codice nel pacchetto può interferire con l'implementazione della superclasse.
- I membri protetti sono introdotti per consentire implementazioni efficienti delle sottoclassi. Possono esistere variabili di istanza protette, oppure le variabili di istanza possono essere private, con accesso tramite metodi protetti. Quest'ultimo approccio è utile se consente alla superclasse di mantenere un invariante significativo.

7.7 - Interfacce

Una classe viene utilizzata per definire un tipo e anche per fornire un'implementazione completa o parziale. Un'interfaccia, invece, **definisce solo un tipo**. Contiene solo **metodi pubblici** non statici e tutti i suoi metodi sono **astratti**. Non fornisce alcuna implementazione. Viene invece implementata da una classe che ha una clausola `implements` nella sua intestazione.

Per esempio, l'interfaccia che definisce il tipo `Iterator`:

```
java
public interface Iterator {

    public boolean hasNext();
    // EFFECTS: Ritorna true se ci sono più oggetti da produrre,
    // altrimenti restituisce false

    public Object next() throws NoSuchElementException;
    // MODIFIES: this
    // EFFECTS: Se non ci sono altri elementi da produrre, lancia
    // NoSuchElementException. Altrimenti restituisce l'elemento
    // successivo e modifica lo stato di questo per riflettere la
    // restituzione.
}
```

Trattandosi di un'interfaccia, non è necessario dichiarare che i suoi metodi sono pubblici; tuttavia,

continueremo a dichiarare i metodi come pubblici per convenzione.

Oltre a essere più convenienti quando tutti i metodi sono astratti, le interfacce forniscono anche un modo per definire tipi che hanno più supertipi. Una classe può estendere una sola classe, ma può anche implementare una o più interfacce.

Ad esempio, un `SortedIntSet` potrebbe implementare un'interfaccia `SortedCollection`:

```
java
public class SortedIntSet extends IntSet
    implements SortedCollection { ... }
```

`SortedIntSet` è un sottotipo di `IntSet` e `SortedCollection`.

7.8 - Implementazioni multiple

La gerarchia può essere usata per fornire più implementazioni di un tipo. Questo uso può essere pensato come la definizione di una famiglia di tipi molto vincolata, in cui tutti i membri hanno esattamente gli stessi metodi e comportamenti.

Ad esempio, potrebbero esistere implementazioni sia rade che dense dei polinomi. Inoltre, in un programma che utilizza polinomi, potrebbe essere auspicabile utilizzare entrambe le implementazioni. In questo modo, ogni `Poly` può essere rappresentato nel modo migliore.

Tuttavia, si desidera considerare gli oggetti di diverse implementazioni come appartenenti allo stesso tipo.

Quando si usa l'ereditarietà per fornire più implementazioni, il tipo da implementare sarà definito da un'interfaccia o da una classe astratta: lo scopo è quello di rimandare i dettagli dell'implementazione alle sottoclassi. Inoltre, le sottoclassi forniranno esattamente il comportamento definito dalla specifica dell'interfaccia o della classe astratta, con l'eccezione che devono fornire dei costruttori.

Le sottoclassi di implementazione sono in gran parte invisibili agli utenti. L'unico punto in cui gli utenti devono conoscerle è quando creano nuovi oggetti. A quel punto, il codice deve chiamare il costruttore della sottoclasse appropriata.

Ad esempio, il programmatore di codice che utilizza `Poly` deve decidere se creare un `Poly` rado o denso ogni volta che viene creato un nuovo oggetto `Poly`.

7.8.1 - Liste

Come primo esempio, consideriamo l'astrazione `IntList`:

```
public abstract class IntList {
    // OVERVIEW: IntLists sono liste immutabili di Objects.
    // Una IntList tipica è la sequenza [x1, ..., xn].
```

```

// methods
public abstract Object first() throws EmptyException;
    // EFFECTS: Se this è vuoto lancia EmptyException altrimenti
    // ritorna il primo elemento di this.

public abstract IntList rest() throws EmptyException;
    // EFFECTS: Se questo è vuoto lancia EmptyException altrimenti
    // restituisce l'elenco contenente tutti gli elementi di questo,
    // tranne il primo, nell'ordine originale.

public abstract Iterator elements();
    // EFFECTS: Ritorna un generatore che produrrà gli elementi di
    // this, ciascuno esattamente una volta, nel loro ordine in this.

public abstract IntList addEl(Object x);
    // EFFECTS: Aggiunge x all'inizio di this.

public abstract int size();
    // EFFECTS: Restituisce un conteggio del numero di elementi di this.

public abstract boolean repOk();
public String toString()
public boolean equals(IntList o)
}

```

In questo caso, il tipo in cima alla gerarchia è definito da una classe astratta:

```

public abstract class IntList {
    // OVERVIEW: IntLists sono liste immutabili di Objects.
    // Una IntList tipica è la sequenza [x1, ..., xn].

    // abstract methods
    public abstract Object first() throws EmptyException;
    public abstract IntList rest() throws EmptyException;
    public abstract Iterator elements();
    public abstract IntList addEl(Object x);
    public abstract int size();
    public abstract boolean repOk();

    // methods
    public String toString() { ... }
    public boolean equals(Object o) {
        try { return equals((IntList) o); }
        catch (ClassCastException e) { return false; }
    }
}

```

```

    }

    public boolean equals(IntList o) {
        // compare elements using elements iterator
    }
}

```

Non ha variabili di istanza e non c'è un costruttore, poiché non esiste un rappresentante. `toString` e `equals` sono implementati utilizzando l'iteratore `elements`. Vengono fornite due definizioni per `equals`: per migliorare le prestazioni nel caso comune in cui il codice chiamante stia verificando l'uguaglianza di due oggetti `IntList`. Questo è simile all'*overloading* del metodo `subset` per `IntSet`.

Alcune parti dell'implementazione dell'elenco vuoto e dell'elenco non vuoto:

```

public class EmptyIntList extends IntList {
    public EmptyIntList() { };

    public Object first() throws EmptyException {
        throw new EmptyException("EmptyIntList.first");
    }
    public IntList addEl(Object x) { return new FullIntList(x); }
    public boolean repOk() { return true; }
    public String toString() { return "IntList: [ ]"; }
    public boolean equals (Object x) { return (x instanceof EmptyIntList); }
    // implementations of rest and size go here
    public Iterator elements() { return new EmptyGen( ); }

    static private class EmptyGen implements Iterator {
        EmptyGen() {}
        public boolean hasNext() { return false; }
        public Object next() throws NoSuchElementException {
            throw new NoSuchElementException("IntList.elements");
        }
    } // end EmptyGen
}

public class FullIntList extends IntList {
    private int sz;
    private Object val;
    private IntList next;

    public FullIntList (Object x) {
        sz = 1; val = x; next = new EmptyIntList( );
    }
}

```

```

    }

    public Object first() { return val; }
    public Object rest() { return next; }
    public IntList addEl(Object x) {
        FullIntList n = new FullIntList(x);
        n.next = this;
        n.sz = this.sz + 1;
        return n; }
    // implementations of elements, size, repOk go here
}

```

Nell'implementazione dell'elenco vuoto non sono necessarie variabili di istanza, quindi si risparmia spazio. Inoltre, le implementazioni dei metodi sono più efficienti di quanto sarebbe possibile senza implementazioni multiple, poiché si evitano i test; ad esempio, `First` lancia sempre un'eccezione in `EmptyIntList` e non lancia mai un'eccezione in `FullIntList`.

Queste implementazioni illustrano un punto importante sull'uso della gerarchia per ottenere implementazioni multiple: le **sottoclassi nella gerarchia possono non essere indipendenti l'una dall'altra**. Così, `EmptyIntList` usa il costruttore di `FullIntList` e `FullIntList` usa il costruttore di `EmptyIntList` nel suo costruttore. Questo è diverso dalle gerarchie che forniscono un **comportamento esteso**; in questo caso, le sottoclassi possono essere implementate indipendentemente l'una dall'altra.

Un altro punto è che non è utile per questi sottotipi fornire una definizione sovraccaricata di `equals`; cioè, non ha senso fornire `EmptyIntList`:

```
public boolean equals(EmptyIntList x)
```

Il punto centrale delle implementazioni multiple è che l'uso del codice è scritto interamente in termini di supertipo, tranne che per la creazione di oggetti. Pertanto, una definizione sovraccaricata di `equals` non verrebbe mai richiamata.

Un problema delle implementazioni mostrate è che ci saranno molti oggetti elenco vuoti. Questo non è necessario, poiché tutti gli oggetti elenco vuoti sono esattamente uguali e gli elenchi sono immutabili. Se si può utilizzare un solo oggetto elenco vuoto, si migliorano le prestazioni evitando la creazione e la successiva *garbage collection* degli oggetti elenco vuoti in più.

7.8.2 - Polinomi

Come secondo esempio, si consideri il tipo `Poly` e si supponga di voler fornire implementazioni

diverse per polinomi radi e densi. Utilizzeremo la classe astratta:

```
public abstract class Poly {
    protected int deg; // il grado

    // constructor
    protected Poly(int n) { deg = n; }

    // abstract methods coeff, repOk, add, mul, minus, terms

    // methods
    public int degree() { return deg; }
    public boolean equals(Object o) {
        try { return equals((Poly) o); }
        catch (ClassCastException e) { return false; }
    }
    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;
    }
    public sub(Poly p) { return add(p.minus( )); }
    public String toString( ) { ... }
}
```

Sebbene la maggior parte dei metodi sia astratta, alcuni non lo sono. Abbiamo scelto di mantenere il grado come variabile di istanza di `Poly`, poiché si tratta di un'informazione utile per tutte le sottoclassi di `Poly`. Inoltre, abbiamo reso `deg` protetto, in modo che le sottoclassi possano accedervi direttamente, anche se abbiamo previsto un costruttore per inizializzare `deg`. Forniamo l'accesso diretto a `deg` perché `Poly` non è in grado di preservare da solo alcuna variante interessante. L'implementazione di `DensePoly` è simile a quanto visto in precedenza compreso l'uso di `deg`. La differenza principale è che non è necessario implementare i metodi forniti dalla superclasse:

```
public class DensePoly extends Poly {
    private int[] trms; // coefficienti fino al grado

    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]; }

// implementazioni di coeff, add, mul, minus,
// terms, and repOk vanno qua

public Poly add(Poly q) throws NullPointerException {
    if (q instanceof SparsePoly) return q.add(p);
    DensePoly la, sm;
    if (deg > q.deg) {la = this; sm = (DensePoly) q;}
    else {la = (DensePoly) q; sm = this;}
    int newdeg = la.deg; // il nuovo grado è il grado più grande
    if (sm.deg == la.deg) // a meno che non ci siano zeri finali
        for (int k = sm.deg; k > 0; k--)
            if (sm.trms[k] + la.trms[k] != 0) break; else newdeg--;
    DensePoly r = new DensePoly(newdeg); // prendere un nuovo DensePoly
    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;
}
}

```

Poiché lo scopo della gerarchia in questo esempio è quello di fornire implementazioni efficienti per gli oggetti `Poly`, dobbiamo decidere all'interno delle implementazioni dei vari metodi `Poly`, come `add`, se il nuovo oggetto `Poly` deve essere denso o rado. Questo requisito complica l'implementazione. Ad esempio, il metodo `add` di `DensePoly` permette a `SparsePoly` di gestire il caso dell'addizione di un `Poly` rado e di un `Poly` denso; altrimenti, il risultato è un `Poly` denso. Tuttavia, quest'ultima decisione potrebbe essere sbagliata, poiché l'addizione potrebbe introdurre molti zeri intermedi. Pertanto, al termine di `add`, si potrebbe verificare questa condizione e convertire in un oggetto `SparsePoly`, se necessario. La decisione potrebbe basarsi sul numero di coefficienti non nulli rispetto al grado. Allo stesso modo, nell'implementazione rada, potremmo convertire nella rappresentazione densa se la nuova `Poly` ha molti coefficienti non nulli per i termini inferiori al grado. Queste conversioni significano che i `DensePoly` devono avere un modo efficiente per creare gli `SparsePoly` e viceversa. Per esempio, potremmo fornire:

```

SparsePoly(int[] trms)
    // EFFECTS: Inizializza "this" come lo stesso polinomio rappresentato da
    // trms nell'implementazione di DensePoly.

```

Ovviamente, un tale costruttore non può essere pubblico! Renderlo protetto non funzionerebbe,

perché `DensePoly` non è una sottoclasse di `SparsePoly`. Invece, il costruttore è visibile dal pacchetto, il che significa che dobbiamo collocare tutte le implementazioni nello **stesso pacchetto**, in modo che abbiano abilità simili per accedere l'una all'altra implementazione. Quando più implementazioni sono scritte con conoscenza reciproca, questa è una cosa ragionevole da fare.

Un'ultima osservazione: sarebbe bello se gli utenti potessero ignorare del tutto la distinzione tra rappresentazioni dense e rade. In questo caso, però, gli utenti avranno bisogno di un modo generico per creare nuovi monomi. Questo può essere fornito da un'altra classe contenente metodi statici che possono essere usati per creare oggetti, ad esempio:

```
public class polyProcs {
    public static Poly makePoly()
        // EFFECTS: Ritorna un Poly zero.

    public static Poly makePoly(int c, int n) throws NegExpException
        // effects: Se n < 0 lancia NegExpException altrimenti ritorna
        // il monomio cx^n.
}
```

Tale tecnica prende il nome di *factory pattern*.

Il primo metodo `makePoly` restituisce un `DensePoly`; il secondo sceglie tra una rappresentazione rada e densa in base al valore di `n`.

7.9 - Il significato dei sottotipi

I sottotipi devono soddisfare il **principio di sostituzione**, in modo che gli utenti possano scrivere e ragionare sul codice usando solo la specifica del supertipo. Quando il codice viene eseguito, gli oggetti che utilizza possono appartenere a sottotipi; tuttavia, vogliamo che il codice si comporti come se avesse utilizzato oggetti di tipo supertipo e che i ragionamenti basati sulla specifica del supertipo siano ancora validi.

Pertanto, il principio di sostituzione richiede che la specifica del sottotipo supporti il ragionamento basato sulla specifica del supertipo. Devono essere supportate tre proprietà:

- **Regola della firma:** assicura che se un programma è corretto in base alla **specificità del supertipo**, lo è anche rispetto alla specifica del sottotipo. Gli oggetti sottotipo devono avere tutti i **metodi del supertipo** e le firme dei metodi del sottotipo devono essere **compatibili** con le firme dei corrispondenti metodi del supertipo.
- **Regola dei metodi:** assicura che i ragionamenti sulle chiamate ai metodi di un supertipo siano validi anche se le chiamate vanno effettivamente al codice che implementa un sottotipo. Le chiamate a questi metodi di sottotipo devono **"comportarsi come"** le chiamate ai corrispondenti metodi di supertipo.

- **Regola delle proprietà:** garantisce che il ragionamento sulle proprietà degli oggetti basato sulla specifica del supertipo sia ancora valido quando gli oggetti appartengono a un sottotipo. Il sottotipo deve **conservare tutte le proprietà** che possono essere dimostrate sugli oggetti del supertipo. Le proprietà devono essere indicate nella sezione panoramica della specifica del supertipo.

Tutte queste regole riguardano **solo le specifiche**: ci interessa sapere se le specifiche del supertipo e del sottotipo sono sufficientemente simili da soddisfare il principio di sostituzione.

La regola della firma garantisce che ogni chiamata corretta per il tipo, secondo la definizione del supertipo, sia corretta anche per il sottotipo. Questo requisito è applicato dal compilatore Java. In Java, il sottotipo deve avere tutti i metodi del supertipo, con firme identiche, tranne che per il fatto che un metodo del sottotipo può avere meno eccezioni del corrispondente metodo del supertipo. La regola sulle eccezioni ha senso: il codice scritto in termini di supertipo può gestire le eccezioni elencate nell'intestazione del metodo nel supertipo, ma funzionerà anche in modo corretto se tali eccezioni non vengono lanciate.

La nozione di compatibilità di Java è un po' più **rigida** del necessario: Java richiede che il tipo di ritorno del sottotipo e del supermetodo siano identici, quando in realtà non ci sarebbero problemi di tipo se il metodo del sottotipo restituisse un sottotipo del metodo del supertipo.

Per esempio, sarebbe bello se il tipo `Foo` avesse:

```
java
Foo clone()
```

poiché in questo modo il risultato può essere utilizzato senza calchi, ad esempio,

```
java
Foo x = y.clone()
```

Tuttavia, Java richiede che il `clone` abbia la firma:

```
java
Object clone()
```

che porta all'utilizzo di codice che deve eseguire il cast del risultato restituito da `clone`, ad esempio:

```
java
Foo x = (Foo) y.clone();
```

Gli altri due requisiti garantiscono che gli oggetti sottotipo si comportino in modo abbastanza simile agli oggetti supertipo, in modo che il codice scritto in termini di supertipo non si accorga della

differenza. Questi requisiti **non possono** essere controllati da un compilatore, poiché richiedono un ragionamento sul significato delle specifiche.

7.9.1 - La regola dei metodi

La regola dei metodi riguarda le chiamate ai metodi definiti dal supertipo. Naturalmente, quando gli oggetti interessati appartengono a sottotipi, le chiamate vanno effettivamente al codice fornito dall'implementazione del sottotipo. La regola dice che si può comunque ragionare sul significato di queste chiamate utilizzando le **specifiche del supertipo**, anche se il codice del sottotipo è in esecuzione.

Alcuni esempi di questo tipo di ragionamento:

- Per qualsiasi `IntSet`, se chiamiamo `y.insert(x)`, sappiamo che `x` è nell'insieme quando la chiamata ritorna.
- Per qualsiasi `Poly`, se una chiamata `p.coeff(3)` restituisce `6`, sappiamo che il grado di `p` è almeno `3`.

Tutti gli esempi forniti finora hanno rispettato questo requisito. Infatti, i nostri metodi di sottotipo hanno tutti esattamente la stessa specifica del metodo del supertipo corrispondente, con un'unica eccezione, il metodo `elements` di `SortedIntSet`.

Ogni volta che un metodo viene riformulato, c'è la possibilità di fare cose sbagliate.

Nel caso di `elements`, il nuovo comportamento è accettabile perché abbiamo sfruttato il non-determinismo nella specifica del metodo `elements` di `IntSet`: la sua specifica consente vari ordini per la produzione degli elementi, e uno di questi ordini è l'ordinamento prodotto dagli elementi di `SortedIntSet`.

Quando diamo nuove specifiche per i metodi dei supertipi nei sottotipi, spesso sfruttiamo il non-determinismo in questo modo.

Per capire meglio come la specifica di un metodo di sottotipo possa differire da quella del corrispondente metodo di supertipo, dobbiamo considerare le **precondizioni** e le **postcondizioni**. La precondizione, definita dalla clausola `requires`, è ciò che deve essere garantito dal chiamante per poter effettuare la chiamata. La postcondizione, definita dalla clausola `effects`, è ciò che viene garantito dopo la chiamata (supponendo che la precondizione fosse valida al momento della chiamata).

Un metodo sottotipo può **indebolire** la precondizione e può **rafforzare** la postcondizione:

- **Regola precondizione:** $\text{pre}_{\{\text{super}\}} \Rightarrow \text{pre}_{\{\text{sub}\}}$
- **Regola postcondizione:** $(\text{pre}_{\{\text{super}\}} \ \&\& \ \text{post}_{\{\text{sub}\}}) \Rightarrow \text{post}_{\{\text{super}\}}$

Entrambe le condizioni devono essere soddisfatte per ottenere la compatibilità tra i metodi del sottotipo e del supertipo.

Indebolire la precondizione significa che il metodo del sottotipo richiede al suo chiamante meno di quanto faccia il metodo del supertipo. Questa regola ha senso perché quando il codice è scritto in

termini di specifica del supertipo, deve soddisfare la preconditione del metodo del supertipo. Poiché questa preconditione implica quella del sottotipo, possiamo essere sicuri che la chiamata al metodo del sottotipo sarà legale se la chiamata al metodo del supertipo è legale.

Ad esempio, supponiamo di aver definito il seguente metodo `IntSet` :

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

In un sottotipo di `IntSet` , si potrebbe ridefinire il metodo con la seguente specifica:

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

La definizione di sottotipo soddisfa la regola della preconditione perché ha una preconditione più debole.

Il solo soddisfacimento della regola della preconditione non è sufficiente per la correttezza della specifica del metodo del sottotipo, poiché è necessario tenere in considerazione anche l'effetto della chiamata. Questo aspetto è catturato dalla **regola della postcondizione**.

Questa regola dice che il metodo sottotipo fornisce più del metodo supertipo: quando ritorna, tutto ciò che il metodo supertipo fornirebbe è assicurato, e forse anche alcuni effetti aggiuntivi. Questa regola ha senso perché il codice chiamante dipende dalla postcondizione del metodo del supertipo, ma ciò deriva dalla postcondizione del metodo del sottotipo. Tuttavia, il codice chiamante dipende dalla postcondizione del metodo solo se la chiamata soddisfa la preconditione (perché altrimenti il metodo può fare qualsiasi cosa); per questo motivo la regola è formulata così.

Ad esempio, la definizione del sottotipo `addZero` data prima soddisfa la regola della postcondizione, poiché la sua postcondizione è identica a quella del metodo del supertipo. Tuttavia, anche la seguente definizione di `addZero` sarebbe legale:

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

Nota bene: Se la chiamata soddisfa la preconditione del metodo del supertipo, l'effetto del metodo del sottotipo è quello atteso; se la chiamata non soddisfa la preconditione, allora potrebbe accadere di tutto.

La definizione dell'iteratore di `elements` in `SortedIntSet` rafforza la postcondizione. In

questo caso, entrambi i metodi hanno la stessa preconditione, cioè `true`, il che significa che tutte le chiamate sono legali. La postcondizione del sottotipo promette un ordine ordinato; da questo si può dedurre l'ordine arbitrario indicato nelle specifiche del metodo del supertipo.

Un altro esempio è il seguente. Supponiamo di definire un sottotipo di `IntSet` che, oltre a tenere traccia dei membri attuali dell'insieme, tenga anche un registro di tutti gli elementi che sono mai stati nell'insieme. La sezione panoramica potrebbe dire:

```
java
// OVERVIEW: Un LogIntSet è un IntSet più un log. Anche il log è un
// insieme; contiene tutti i numeri interi che sono stati membri
// dell'insieme.
```

Ecco le specifiche di `insert` per `LogIntSet`:

```
java
public void insert(int x)
    // MODIFIES: this
    // EFFECTS: Aggiunge x all'insieme e anche a log
```

Questo metodo è legale perché la sua postcondizione implica quella del supertipo metodo `insert`: aggiunge `x` all'insieme, ma fa anche qualcos'altro.

Tuttavia, supponiamo di aver definito un sottotipo di `IntSet` in cui abbiamo ridefinito `insert`:

```
java
public void insert(int x)
    // MODIFIES: this
    // EFFECTS: Se x è dispari lo aggiunge a this altrimenti non fa nulla
```

In questo caso, abbiamo violato il requisito; chiaramente questa postcondizione **non implica** quella del metodo `insert` di `IntSet`. Inoltre, un programma scritto in base alle specifiche di `IntSet` si aspetterebbe chiaramente che all'insieme vengano aggiunti numeri pari e dispari!

Un altro esempio di metodo di sottotipo illegale è il seguente. `OrderedIntList` ha un metodo `addEl`:

```
java
public void addEl(int x) throws DuplicateException
    // MODIFIES: this
    // EFFECTS: Se x è in this lancia DuplicateException altrimenti
    // aggiunge x a this
```

Supponiamo di aver definito un sottotipo di `OrderedIntList` in cui il metodo `addEl` non lancia

l'eccezione:

```
java
public void addEl(int x)
    // MODIFIES: this
    // EFFECTS: Se x non è in this lo aggiunge a this
```

Questo metodo soddisfa la regola della firma, perché è consentito che il metodo del sottotipo lanci meno eccezioni rispetto alla specifica del supertipo. Tuttavia, non soddisfa la regola dei metodi perché la regola della postcondizione non è soddisfatta: i due metodi hanno un comportamento diverso nel caso in cui `x` sia già presente nell'elenco.

Un esempio di un caso in cui non lanciare l'eccezione è accettabile è il generatore `allPrimes`. Il metodo `next` di `Iterator` lancia `NoSuchElementException` se non ci sono più elementi. Tuttavia, il metodo `next` del generatore `allPrimes` non lancia l'eccezione; ciò è consentito perché c'è sempre un primo più grande da produrre.

Come ultimo esempio, consideriamo l'`int` e il `long`. Gli `int` sono a 32 bit, mentre i `long` sono a 64 bit. Inoltre, i due tipi hanno comportamenti diversi in alcuni casi. Ad esempio, se l'addizione di due `int` dà luogo a un overflow, l'overflow non si verifica se gli stessi due valori sono `long`. Pertanto `int` non è un sottotipo di `long`, né `long` è un sottotipo di `int`.

7.9.2 - La regola delle proprietà

Oltre a ragionare sugli effetti delle singole chiamate, ragioniamo anche sulle proprietà degli oggetti. Alcune proprietà sono **invarianti**: sono sempre vere per gli oggetti del tipo.

Ad esempio, la dimensione di un `IntSet` è sempre maggiore o uguale a zero.

Altre sono **proprietà evolutive**; si tratta di ragionare su come gli oggetti si evolvono nel tempo.

Ad esempio, se sappiamo che un polinomio ha grado 6, sappiamo che avrà sempre questo grado (poiché i polinomi sono immutabili).

Per dimostrare che un sottotipo soddisfa la regola delle proprietà, dobbiamo provare che conserva ogni proprietà del supertipo. Nel caso di una proprietà invariante, si fa il normale tipo di prova usando l'induzione dei tipi di dati: i creatori e i produttori del sottotipo devono stabilire l'invariante e tutti i metodi del sottotipo devono preservare l'invariante.

Nota bene: ora ci occupiamo dei metodi "extra" e di quelli ereditati: *tutti i* metodi devono preservare l'invariante. Inoltre, dobbiamo considerare i costruttori di sottotipi e assicurarci che stabiliscano l'invariante.

Nel caso di una proprietà evolutiva, dobbiamo dimostrare che ogni metodo la preserva.

Per esempio, supponiamo di voler dimostrare che il grado di un `Poly` non cambia. Prima di considerare i sottotipi, il modo per dimostrarlo è assumere che il grado di un oggetto `Poly p` sia un certo valore `x`, e poi sostenere che ogni metodo `Poly` non cambia questo valore. Con i sottotipi, dobbiamo fare lo stesso ragionamento per tutti i metodi del sottotipo, ad esempio per tutti i metodi `DensePoly` e tutti i metodi `SparsePoly`.

Le proprietà di interesse devono essere definite nella sezione `OVERVIEW` della specifica del supertipo. Le proprietà invarianti derivano dal **modello astratto**.

Ad esempio, poiché gli `IntSet` sono modellati come insiemi matematici, devono avere una dimensione maggiore o uguale a zero e non devono contenere elementi duplicati. Inoltre, poiché le `OrderedIntList` sono modellate come sequenze ordinate in ordine crescente, sappiamo che i loro elementi appaiono in ordine ordinato.

Come altro esempio di proprietà invariante, si consideri un tipo `FatSet` i cui oggetti non sono mai vuoti. Questo fatto dovrebbe essere catturato nella sezione dell'`OVERVIEW`:

```
java
// OVERVIEW: Un FatSet è un insieme mutabile di numeri interi la cui
// dimensione è sempre almeno pari a 1.
```

Si supponga che `FatSet` non abbia un metodo `remove`, ma abbia invece un metodo `removeNonEmpty`:

```
java
public void removeNonEmpty(int x)
    // MODIFIES: this
    // EFFECTS: Se x è in this e this contiene altri elementi rimuove x
    // da this
```

e, inoltre, che ogni costruttore di `FatSet` crea un insieme contenente almeno un elemento. Pertanto, possiamo dimostrare che gli oggetti `FatSet` hanno dimensioni maggiori di zero. Si consideri ora `ThinSet`, che ha tutti i metodi di `FatSet` con specifiche identiche, in più:

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

`ThinSet` non è un sottotipo legale di `FatSet` perché il suo metodo aggiuntivo può far sì che il suo oggetto diventi vuoto; pertanto, non conserva l'invariante del supertipo.

L'unica proprietà evolutiva che abbiamo visto finora (e la più comune) è l'**immutabilità**.

Consideriamo un tipo `SimpleSet` che ha solo i metodi `insert` e `isIn`, in modo che gli oggetti `SimpleSet` crescano soltanto. Questo fatto deve essere indicato nell'`OVERVIEW`:


```
java
// OVERVIEW: Un SimpleSet è un insieme mutabile di numeri interi.
// Gli oggetti SimpleSet possono crescere nel tempo, ma non ridursi.
```

`IntSet` non può essere un sottotipo di `SimpleSet` perché il suo metodo `remove` causa la contrazione degli insiemi.

Un tipo immutabile di solito avrà solo **sottotipi immutabili**, ma questo non è un requisito. Dire che un tipo è immutabile significa che le mutazioni non possono essere osservate usando i metodi del supertipo.

Per esempio, supponiamo che il tipo `Point2` rappresenti i punti del piano e che la sezione `OVERVIEW` affermi che gli oggetti `Point2` sono immutabili. `Point2` potrebbe avere un sottotipo i cui oggetti sono linee nel piano, costituite da un punto nel piano e da un angolo. Questo sarebbe un sottotipo legittimo anche se fornisse un metodo che permetta di cambiare l'angolo, poiché tale cambiamento non sarebbe visibile tramite chiamate ai metodi del supertipo.

7.9.3 - Uguaglianza

Precedentemente abbiamo discusso il significato del metodo `equals`: se due oggetti sono uguali, non sarà mai possibile distinguerli in futuro usando i metodi del loro tipo. Come già discusso, ciò significa che per i tipi mutabili gli oggetti sono uguali solo se sono lo stesso oggetto, mentre per i tipi immutabili sono uguali se hanno lo stesso stato.

Quando ci sono sottotipi di tipi immutabili, gli oggetti del sottotipo potrebbero avere più stati o potrebbero anche essere mutabili. Pertanto, gli oggetti sottotipo potrebbero essere distinguibili, anche se il codice che li utilizza tramite l'interfaccia del supertipo non può distinguerli.

Ad esempio, si consideri un tipo, `Point2`, che rappresenta punti nello spazio a due coordinate; il suo metodo `equals` restituisce `true` se le coordinate `x` e `y` sono uguali. Supponiamo ora che il tipo `Point3`, che rappresenta punti nello spazio a tre, sia definito come un sottotipo di `Point2`; il metodo `equals` di `Point3` restituirà `true` solo se tutte e tre le coordinate sono uguali. Per implementare correttamente questo comportamento, `Point3` deve fornire il proprio metodo `equals` aggiuntivo e deve sovrascrivere anche `equals` per `Point2` e `Object`:

```
java
public class Point3 extends Point2 {
    private int z; // la coordinata z

    public boolean equals(Object p) { // definizione sovrascritta
        if (p instanceof Point3) return equals((Point3) p);
        return super.equals(p);
    }
}
```

```

    public boolean equals(Point2 p) { // definizione sovrascritta
        if (p instanceof Point3) return equals((Point3) p);
        return super.equals(p);
    }

    public boolean equals(Point3 p) { // definizione extra
        if (p == null || z != p.z) return false;
        return super.equals(p);
    }
}

```

La sovrascrittura di questi metodi assicura che `equals` funzioni correttamente sugli oggetti `Point3`, indipendentemente dal loro tipo apparente.

7.10 - Discussione sulla gerarchia di tipi

Esistono tre tipi diversi di supertipi:

- **Supertipi incompleti:** stabiliscono convenzioni di denominazione per i metodi dei sottotipi, ma non forniscono specifiche utili per tali metodi. Pertanto, l'uso del codice non è tipicamente scritto in termini di questi metodi.

Per esempio, supponiamo di voler definire una serie di tipi di collezioni, in modo che metodi simili abbiano nomi simili. Alcuni sottotipi di collezione potrebbero essere mutabili, mentre altri no. Il supertipo definirebbe sia osservatori che mutatori, ma ovviamente i mutatori non farebbero nulla per i sottotipi immutabili. Per esempio, potremmo avere:

```

java
public void put(Object x) throws UnsupportedOperationException
    // MODIFIES: this
    // EFFECTS: Se this è mutabile aggiunge x a this altrimenti
    // lancia UnsupportedOperationException

```

Questo supertipo è inutile per quanto riguarda l'uso del codice, o almeno per il codice che usa i mutatori. Tuttavia, il supertipo serve a standardizzare i nomi dei metodi, in modo che tutti i sottotipi di collezione che sono mutabili abbiano un metodo chiamato `put` che aggiunge il suo argomento alla collezione. I tipi di collezione in `java.util` sono definiti in questo modo.

- **Supertipi completi:** forniscono intere astrazioni di dati, con utili specifiche per tutti i metodi.

Un esempio è `Reader`.

- **Frammenti** (*Snippets*): fornisce solo alcuni metodi, non sufficienti per essere considerati un'intera astrazione di dati. Tuttavia, questi metodi sono specificati in modo tale da permettere di scrivere il codice d'uso in termini di supertipo.

Ad esempio, `Cloneable` è un frammento: indica solo che i sottotipi hanno un metodo `clone`. Gli *snippet* sono sempre definiti da interfacce.

Indipendentemente dal tipo di supertipo, tuttavia, i sottotipi devono soddisfare il **principio di sostituzione**. Quando i supertipi non sono completi, questo può essere facile.

Per esempio, i supertipi che definiscono gli *snippet* hanno in genere solo uno o due metodi e nessuna proprietà che i sottotipi devono preservare. Inoltre, i supertipi incompleti di solito non hanno proprietà, perché lo scopo è lasciare i dettagli ai sottotipi.

Il principio di sostituzione preclude **l'uso dell'ereditarietà** come semplice meccanismo di condivisione del codice, perché richiede che il sottotipo sia simile al supertipo. È importante tenerlo a mente perché molti linguaggi di programmazione (non Java) incoraggiano un uso improprio dell'ereditarietà, ad esempio consentendo agli oggetti della sottoclasse di non fornire alcuni metodi della superclasse.

Pertanto, i vantaggi della gerarchia sono:

- La gerarchia può essere utilizzata per definire la **relazione tra un gruppo di tipi**, rendendo più facile la comprensione del gruppo nel suo complesso.
- La gerarchia consente di scrivere il codice in termini di un **supertipo**, per poi lavorare per molti tipi, tutti i sottotipi di quel supertipo.
- La gerarchia fornisce **estensibilità**: il codice può essere scritto in termini di un supertipo, ma può continuare a funzionare anche quando i sottotipi vengono definiti in seguito.
- Tutti questi vantaggi possono essere ottenuti *solo* se i sottotipi rispettano il **principio di sostituzione**.

8 - Astrazioni polimorfiche

Dover definire una nuova versione di un'astrazione di collezione ogni volta che si ha bisogno di memorizzare un tipo diverso di elemento non è molto soddisfacente. Sarebbe invece meglio definire il tipo di collezione una sola volta e farla funzionare per tutti i tipi di elementi.

Questo obiettivo può essere raggiunto definendo **astrazioni polimorfiche**. Queste astrazioni sono chiamate **polimorfiche** perché funzionano per molti tipi.

Il **polimorfismo** generalizza le astrazioni in modo che funzionino per **molti tipi**. Ci permette di evitare di dover ridefinire le astrazioni quando vogliamo usarle per più tipi; al contrario, una singola astrazione diventa molto più utile.

Ad esempio, l'astrazione `Vector` è polimorfica rispetto al tipo di elemento.

Una procedura o un iteratore possono essere polimorfici rispetto ai tipi di uno o più argomenti.

Un'astrazione di dati può essere polimorfica rispetto ai tipi di elementi contenuti nei suoi oggetti.

Ad esempio, si potrebbe definire una routine per rimuovere un elemento di tipo arbitrario da un vettore.

Le astrazioni polimorfiche sono auspicabili perché forniscono un modo per astrarre dai tipi di parametri. In questo modo, possiamo ottenere un'astrazione più potente, che funziona per molti tipi piuttosto che per un singolo tipo. Le procedure, gli iteratori e le astrazioni di dati possono beneficiare di questa tecnica.

In Java, il polimorfismo si esprime attraverso la gerarchia. Alcuni argomenti sono dichiarati come **appartenenti a un supertipo**, mentre gli argomenti effettivi possono essere oggetti appartenenti a sottotipi di quel tipo. Spesso il supertipo è `Object`. In questo caso, l'astrazione polimorfa è limitata all'uso di metodi `Object`, come `equals`, sui suoi parametri. A volte, però, l'astrazione polimorfa ha bisogno di utilizzare metodi aggiuntivi, e in questo caso si sceglie il supertipo per fornire tali metodi.

8.1 - Astrazioni di dati polimorfi

Una specifica dell'astrazione `Set`:

```
public class Set {
    // OVERVIEW: Set sono insiemi di oggetti mutevoli e senza limiti.
    // null non è mai un elemento di un insieme. I metodi utilizzano
    // equals per determinare l'uguaglianza degli elementi.

    // constructors
    public Set()
        // EFFECTS: Inizializza this per essere vuoto
```

```

// methods
public void insert(Object x) throws NullPointerException
    // MODIFIES: this
    // EFFECTS: Se x è null lancia NullPointerException altrimenti
    // aggiunge x agli elementi di this

public void remove(Object x)
    // MODIFIES: this
    // EFFECTS: Se x è in this, rimuove x da this, altrimenti non fa
    // niente

public boolean isIn(Object x)
    // EFFECTS: Ritorna true se x è in this altrimenti false

public boolean subset(Set s)
    // EFFECTS: Se tutti gli elementi di this sono elementi di s
    // ritorna true altrimenti restituisce false specifiche di
    // dimensione ed elementi.
}

```

Gli oggetti `Set` contengono collezioni eterogenee di elementi (simili a `Vector`). La specifica di `Set` è simile a quella di `IntSet`, tranne per il fatto che i suoi metodi (come `insert` e `isIn`) prendono oggetti come argomenti o li restituiscono come risultati. Poiché gli oggetti possono essere confrontati in vari modi (utilizzando `==` o `equals`), la panoramica indica quale test di uguaglianza viene utilizzato.

Poiché `Set` memorizza `Object`, si pone la questione se `null` possa essere un elemento legale. A questa domanda viene data una risposta esplicita (negativa) nella sezione `OVERVIEW` delle specifiche; la restrizione è applicata da `insert`, come indicato nelle sue specifiche.

Un'implementazione parziale di `Set` è:

```

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( )); }
}

```

Questa implementazione differisce poco da quella fornita per `IntSet` e, il suo invariante di rappresentazione e la sua funzione di astrazione sono simili a quelli di `IntSet`:

```

// La funzione di astrazione è:
// AF(c) = c.els[i] | 0 <= i < c.sz

// L'invariante di rappresentazione I(c) è:
// c.els != null && per ogni 0 <= i < c.size (c.els[i] != null) &&
// per ogni 0 <= i < j < c.els.size ( !c.els[i].equals(c.els[j]) )

```

Questa funzione di astrazione produce gli oggetti in `c.els` piuttosto che gli `int` contenuti in tali oggetti. L'invariante di rappresentazione include la condizione che l'insieme non contiene `null`; dipende dal metodo `equals` per determinare l'uguaglianza degli elementi.

Si noti che `insert` memorizza l'oggetto del suo argomento nell'insieme e non un clone dell'oggetto. Questo comportamento è indicato nelle sue specifiche, che dicono che aggiunge `x` all'insieme, cioè proprio quell'oggetto, e non un suo clone; se fosse stato richiesto un clone, le specifiche lo avrebbero detto esplicitamente.

Si noti anche che il metodo `clone` non clona gli elementi dell'insieme, ma clona solo il **vettore** `els`. Pertanto, l'insieme clonato condivide i suoi elementi con l'insieme clonato. Nessuna di queste implementazioni espone la rep, perché lo stato di un insieme, o, in effetti, di quasi tutte le collezioni polimorfiche, consiste solo nelle identità dei suoi elementi e non nei loro stati.

8.2 - Utilizzo delle astrazioni di dati polimorfi

Le astrazioni di dati polimorfiche sono utilizzate in modo simile alle loro controparti non polimorfiche, con due differenze principali.

1. Solo gli oggetti possono essere memorizzati nell'insieme e quindi i valori primitivi come gli `int` devono essere avvolti nel loro tipo di oggetto corrispondente

2. Gli osservatori che restituiscono elementi dell'insieme restituiranno `Object` e quindi il codice che li utilizza dovrà eseguire il `cast` al tipo previsto e, nel caso di un valore primitivo, effettuare l'`unwrap`.

Ad esempio:

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

C'è una differenza importante tra questo codice e quello che utilizza un `IntSet`. Un `IntSet` memorizza solo `int` e questa garanzia è fornita dal compilatore: non è possibile chiamare `insert` su un oggetto `IntSet` passando qualcosa di diverso da un `int` come argomento. Questa garanzia non viene fornita per gli insiemi. Anche se un uso tipico è quello di avere un insieme omogeneo in cui tutti gli elementi sono dello stesso tipo, il compilatore **non applicherà il vincolo**. Ciò significa che è possibile una classe di errori quando si usano collezioni polimorfiche che non si possono verificare quando si usa una collezione specifica come `IntSet`.

8.3 - L'uguaglianza rivisitata

Una collezione come `Set` determina se un elemento è un membro della collezione utilizzando il metodo `equals`.

Pertanto, il contenuto di un oggetto del tipo di collezione dipende da come viene implementato `equals` per gli elementi.

Ad esempio, `equals` per `Vector` restituisce effettivamente `true` se i due vettori hanno lo stesso stato; altri tipi di collezione in `java.util` definiscono `equals` in modo simile. Questo significa che bisogna fare attenzione quando si memorizzano vettori in un `Set`.

Se si usa l'insieme per tenere traccia di oggetti vettoriali distinti, l'implementazione non farà ciò che si desidera. Ad esempio, si consideri il seguente codice:

```
java
Set s = new Set();
Vector x = new Vector();
Vector y = new Vector();
s.insert(x);
s.insert(y); // y non viene aggiunto a s, poiché sembra che sia già
```

```
// presente in esso
x.add(new Integer(3));
if (s.isIn(y)) // non arriverà qua
```

Poiché `y` ha lo stesso stato di `x` quando viene inserito in `s`, sembra essere già in `s` e quindi non viene aggiunto di nuovo. Tuttavia, quando lo stato di `x` cambia, `y` non è più uguale a `x` e quindi la chiamata a `isIn` restituisce `false`.

Potrebbe sembrare che il modo per evitare questo problema sia quello di usare `==` invece di `equals` per confrontare gli elementi, ma questo approccio non funziona correttamente per i tipi immutabili.

Ad esempio, probabilmente non si vuole che l'insieme contenga due copie della stringa `"abc"`! Un modo per risolvere il problema è quello di avvolgere i vettori in oggetti contenitore quando si intende distinguere oggetti vettoriali distinti.

Ad esempio esempio:

```
java
public class Container {
    // OVERVIEW: Un Container contiene un singolo oggetto. Due Container
    // sono equals se contengono lo stesso oggetto.
    // I contenitori sono immutabili

    private Object el;

    // constructor
    public Container(Object x) {
        // EFFECTS: Fa sì che this contenga x
        el = x;
    }

    // methods
    public Object get() {
        // EFFECTS: Ritorna l'oggetto nel contenitore.
        return el;
    }

    public boolean equals(Object x) {
        if (! x instanceof Container) return false;
        return (el == ((Container) x).el);
    }
}
```

Un contenitore è immutabile e due contenitori sono uguali se contengono lo stesso oggetto. Si noti che `Container` è esso stesso polimorfico.

Ora possiamo inserire sia `x` che `y` nell'insieme, anche se hanno lo stesso valore di stato:

```
java
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))) // arriverà qua
```

Questo codice fa sì che `s` contenga due elementi, uno per il vettore `x` e l'altro per il vettore `y`. Pertanto, anche se `x` viene modificato, troviamo ancora `y` nell'insieme. Si noti che ora dobbiamo passare i contenitori come argomenti ai metodi `Set`.

8.4 - Metodi aggiuntivi

Il tipo `Set` e molte altre astrazioni polimorfiche di dati utilizzano solo i metodi `Object` per i loro parametri, ma alcune astrazioni necessitano di metodi aggiuntivi.

Per esempio, supponiamo di voler definire un tipo `OrderedList`, una versione polimorfica di `OrderedList`. Per definire un tale tipo, abbiamo bisogno di un modo per ordinare gli elementi. `Object` non fornisce un modo per farlo. La capacità richiesta può essere ottenuta definendo un supertipo, i cui sottotipi abbiano tutti un metodo di confronto. Un tale tipo, chiamato `Comparable`, è definita in `java.util`:

```
java
public interface Comparable{
    // OVERVIEW: I sottotipi di Comparable forniscono un metodo per
    // determinare l'ordinamento dei loro oggetti. Questo ordinamento
    // deve essere un ordine totale sui loro oggetti e deve essere
    // transitivo e simmetrico.
    // Inoltre 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 comparabili, lancia ClassCastException.
    // In caso contrario, se this è più piccolo di x ritorna -1;
    // se this è uguale a x ritorna 0; e se this è più grande di x,
    // ritorna 1.
}
```

Un punto da notare è che diversi oggetti potrebbero non essere comparabili. `compareTo`

potrebbe essere chiamato con `null` o con un argomento appartenente a un tipo che non è un sottotipo di `Comparable`. Ma anche se l'argomento appartiene a un sottotipo di `Comparable`, può comunque esserci un problema.

Ad esempio, `Integer` e `String` sono sottotipi di `Comparable`, tuttavia `x.compareTo(s)`, dove `x` è un `Integer` e `s` è una `String`, non ha senso. I tentativi di fare confronti di questo tipo causeranno il lancio di `ClassCastException` da parte di `compareTo`.

Data l'interfaccia `Comparable`, possiamo definire `OrderedList`:

```
java
public class OrderedList {
    // OVERVIEW: Un elenco ordinato è un elenco ordinato mutabile di
    // oggetti Comparable. Una lista tipica è una sequenza [x1, ..., xn]
    // where xi < xj se i < j.
    // L'ordinamento degli elementi viene effettuato utilizzando il loro
    // metodo compareTo.

    private boolean empty;
    private OrderedList left, right;
    private Comparable val;

    // constructors
    public OrderedList() {
        // EFFECTS: Inizializza this per essere una lista ordinata vuota
        empty = true;
    }

    // methods
    public void addEl(Comparable el) throws NullPointerException,
        DuplicateException, ClassCastException {
        // MODIFIES: this
        // EFFECTS: Se el è in this, lancia DuplicateException; se el è
        // null lancia NullPointerException; se el non può esser
        // e comparato con altri elementi di this lancia
        // ClassCastException; in caso contrario, 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);
    }

    public void remEl(Comparable el) throws NotFoundException
        // MODIFIES: this
        // EFFECTS: Se el non è in this, lancia NotFoundException;
        // in caso contrario, rimuove el da this.

    public boolean isIn(Comparable el)
        // EFFECTS: Se el è in this ritorna true altrimenti ritorna false.
    }

```

Come nel caso di `Set`, le specifiche e l'implementazione sono simili a quelle del tipo correlato, `OrderedIntList`. Le differenze principali sono: gli argomenti e i risultati sono ora `Comparable`, mentre prima erano `int`, e il confronto viene effettuato utilizzando `compareTo`, come indicato nella panoramica.

`OrderedList` in realtà assicura che gli elementi dell'elenco siano omogenei. Questo accade perché `compareTo` lancia un'eccezione se gli oggetti non sono confrontabili, cioè se non appartengono a tipi correlati per i quali il confronto ha senso. Il tipo di elemento dell'elenco viene determinato quando viene aggiunto il primo elemento; se l'elenco diventa vuoto, questo tipo può passare a qualcosa di diverso quando viene aggiunto l'elemento successivo. Si noti che `addEl` si assicura che il primo elemento sia comparabile, rifiutando il tentativo di aggiungere null all'elenco.

Affinché gli elementi di un tipo possano essere memorizzati in un `OrderedList`, il tipo deve essere un sottotipo di `Comparable`. Ogni tipo per il quale ciò ha senso dovrebbe essere definito in questo modo.

8.5 - Maggiore Flessibilità

L'uso di un supertipo come `Comparable` per catturare i requisiti di un'astrazione polimorfa rispetto ai metodi che utilizza per i suoi parametri richiede una pianificazione preliminare. Il supertipo deve essere definito per primo, prima che vengano definiti i tipi che dovrebbero essere i suoi sottotipi; poi questi tipi possono essere definiti per "implementare" il supertipo. Questa pianificazione preventiva non è sempre possibile. A volte un tipo di collezione è definito *prima di* alcuni tipi di elementi desiderati. In questo caso, abbiamo bisogno di un altro modo per accedere ai metodi utilizzati nella collezione.

Questo si può ottenere definendo un'interfaccia i cui oggetti abbiano i metodi richiesti, ma ora i tipi di

elementi non sono sottotipi di tale interfaccia. Invece, per ogni tipo che sarà usato per gli elementi della collezione, si deve definire un **sottotipo speciale** del tipo di interfaccia.

Ad esempio, supponiamo di volere un insieme che mantenga una somma continua dei suoi elementi. Ogni volta che un elemento viene inserito, la somma viene incrementata; quando un elemento viene rimosso, la somma viene decrementata. Per mantenere la somma, tuttavia, l'insieme deve utilizzare i metodi del suo tipo di elemento: uno per aggiungere e uno per sottrarre. Possiamo catturare questo requisito nell'interfaccia `Adder`:

```
java
public interface Adder {
    // OVERVIEW: Tutti i sottotipi di Adder forniscono un mezzo per
    // aggiungere e sottrarre gli elementi di un tipo di oggetto
    // correlato.

    public Object add(Object x, Object y) throws NullPointerException,
        ClassCastException;
    // EFFECTS: Se x o y sono null, lancia NullPointerException;
    // se x e y sono non sommabili, lancia ClassCastException;
    // altrimenti ritorna la somma di x e y

    public Object sub(Object x, Object y) throws NullPointerException,
        ClassCastException;
    // EFFECTS: Se x o y sono null, lancia NullPointerException;
    // se x e y sono non sommabili, lancia ClassCastException;
    // altrimenti ritorna la differenza di x e y

    public Object zero()
    // EFFECTS: Ritorna l'oggetto che rappresenta lo zero relativo
    // al tipo
}
```

Oltre ai metodi di `add` e `sub`, occorre anche un modo per ottenere l'oggetto `zero` per il tipo di elemento.

L'interfaccia `Adder` non intende essere un supertipo dei tipi i cui elementi possono essere aggiunti. Fornisce invece oggetti i cui metodi possono essere utilizzati per aggiungere o sottrarre elementi di qualche tipo correlato. Per ogni tipo correlato, deve essere definito un sottotipo di `Adder`.

La classe che definisce il sottotipo di `Adder` che aggiunge `Poly`:

```
java
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; }
}

```

Non è necessario fornire una specifica per questo tipo, poiché è solo un'implementazione di `Adder`. In questo caso abbiamo scelto di memorizzare il polinomio zero nella `rep`; in alternativa, potremmo creare il polinomio zero ogni volta che viene chiamato `zero`.

Un punto da notare è che i metodi di `Adder` non sono identici a quelli di del tipo correlato. In questo caso, `Poly` ha metodi denominati `add` e `sub`, ma hanno firme diverse dai metodi `Adder` correlati; inoltre, `Poly` non ha un metodo `zero`.

Come altro esempio, si potrebbe definire una classe `IntegerAdder` che aggiunga `Integers`, anche se `Integers` non hanno metodi aritmetici. `SumSet` è definito in termini di interfaccia `Adder`:

```

java
public class SumSet {
    // OVERVIEW: Gli insiemi di somme sono insiemi mutabili di oggetti più
    // una somma degli oggetti correnti nell'insieme. La somma viene
    // calcolata utilizzando un oggetto Adder. Tutti gli elementi
    // dell'insieme sono sommabili con l'Adder.

    private Vector els; // gli elementi
    private Object s; // la somma degli elementi
    private Adder a; // l'oggetto usato per effettuare la somma
}

```

```

// e la sottrazione

// constructor
public SumSet(Adder p) throws NullPointerException {
    // EFFECTS: Fa sì che this sia l'insieme vuoto i cui elementi
    // possono essere aggiunti usando p, con somma iniziale p.zero
    p.zero.els = new Vector( ); a = p; s = p.zero( ); }

public void insert(Object x) throws NullPointerException,
    ClassCastException {
    // MODIFIES: this
    // EFFECTS: Se x è null lancia NullPointerException; se x non può
    // essere sommato ad altri elementi di this lancia
    // ClassCastException; altrimenti somma x all'insieme e regola
    // 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;
}

```

Gli oggetti `SumSet` sono in realtà omogenei (come `OrderedList`), ma in questo caso il tipo di elemento viene determinato al momento della creazione dell'insieme, tramite l'oggetto `Adder` che è un argomento del costruttore.

Ecco un esempio di utilizzo di questo tipo:

```

java
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();

```

Gli oggetti `SumSet` potranno memorizzare solo oggetti `Poly`.

Un tipo come `SumSet` è un po' scomodo da usare, a causa della necessità di definire un sottotipo di `Adder` per ogni tipo di elemento. Per questo motivo, può essere utile combinare l'uso dell'`Adder` con l'uso di un tipo come `Comparable`.

Ad esempio, potremmo definire un tipo `Addable`, con i seguenti metodi:

```

java
public Object add(Object x) throws NullPointerException,
    ClassCastException
public Object sub(Object x) throws NullPointerException,
    ClassCastException
public Object zero()

```

I tipi di elementi definiti successivamente possono essere definiti come sottotipi di `Addable`. Per esempio, se `Poly` fosse stato definito dopo la definizione di `Addable`, si potrebbe implementare `Addable`, anche se dovrebbe avere metodi aggiuntivi per corrispondere all'interfaccia di `Addable`. Con questo approccio, `SumSet` avrebbe due costruttori:

```

java
public SumSet(Adder p) throws NullPointerException
public SumSet()

```

Il secondo costruttore verrebbe utilizzato per i tipi che sono sottotipi di `Addable`. Alcuni tipi di collezione in `java.util` sono definiti in questo modo. Fanno uso di `Comparable` e anche di un tipo `Comparator`:

```

java
public interface Comparator {
    public int compare(Object x, Object y)
        throws NullPointerException, ClassCastException;
    // EFFECTS: Se x o y è null lancia NullPointerException;
    // se x e y non sono comparabili lancia ClassCastException.
    // In caso contrario, se x è più piccolo di y, returns -1; se
    // x è uguale y, ritorna 0; se x è più grande di y, ritorna 1.
}

```

8.6 - Procedure polimorfiche

Tutti gli esempi fatti finora riguardano astrazioni di dati, ma le stesse tecniche possono essere utilizzate per le procedure.

Ad esempio:

```

java
class Vectors {
    // OVERVIEW: Fornisce procedure utili per la manipolazione dei
    // vettori.

    public static int search(Vector v, Object o)
        throws NotFoundException, NullPointerException
}

```

```

// EFFECTS: Se v è null lancia NullPointerException altrimenti se
// o è in v ritorna l'indice dove o è memorizzato altrimenti
// lancia NotFoundException. Utilizza equal per confrontare o
// con gli elementi di v.

public static sort(Vector v) throws ClassCastException
// MODIFIES: v
// EFFECTS: Se v non è null, lo ordina in ordine crescente usando
// il metodo compareTo di Comparable; se alcuni elementi di v
// sono nulli o non sono confrontabili lancia ClassCastException.

public static sort (Vector v, Comparator c)
    throws ClassCastException
// MODIFIES: v
// EFFECTS: Se v non è null, lo ordina in ordine crescente usando
// il metodo compare di c; se alcuni elementi di v sono nulli o
// non sono confrontabili lancia ClassCastException
}

```

Qui ci sono due definizioni di `sort`: la prima funziona se gli elementi di `v` appartengono a sottotipi di `Comparable`, la seconda prende come argomento un `Comparator`

8.7 - Sintesi

Un'astrazione polimorfa di solito richiede l'accesso a determinati metodi dei suoi parametri. A volte i metodi che tutti gli oggetti hanno, quelli definiti da `Object`, sono sufficienti. Tuttavia, a volte sono necessari altri metodi. In questo caso, l'astrazione polimorfa fa uso di un'interfaccia per definire i metodi necessari.

Esistono due modi diversi di definire questa interfaccia:

1. L'utilizzo dell'interfaccia destinata a essere un **supertipo dei tipi di elementi**. Un esempio è `Comparable`. Questo approccio viene chiamato **sottotipo di elemento**, poiché ogni potenziale tipo di elemento deve essere definito come sottotipo dell'interfaccia. Il problema di questo approccio è che richiede una **pianificazione preliminare**. Se l'astrazione polimorfa viene inventata dopo che alcuni tipi di elementi desiderabili sono già stati definiti, è troppo tardi per rendere tali tipi sottotipi dell'interfaccia.
2. L'utilizzo dell'interfaccia come supertipo di tipi, correlati ai tipi di elementi. Gli oggetti appartenenti a un sottotipo dell'interfaccia hanno metodi che forniscono le funzionalità necessarie agli oggetti del tipo di elemento correlato. `Adder` è un esempio di tale interfaccia. Chiameremo questo approccio **sottotipo correlato**, poiché per ogni tipo di elemento deve essere definito un tipo correlato che sia un sottotipo dell'interfaccia.

L'approccio dei sottotipi correlati è meno conveniente di quello dei sottotipi di elementi, a causa della necessità di definire i sottotipi aggiuntivi. Inoltre, quando i tipi di elemento sono definiti dopo l'astrazione polimorfa, l'approccio del sottotipo di elemento funziona. Pertanto, a volte le astrazioni polimorfiche consentono entrambi gli approcci; il codice d'uso seleziona l'approccio quando costruisce la collezione polimorfa o chiama la procedura polimorfa.