Emiddio Ingenito - Programmazione II - PDJ + EJ + Lezioni Prof. Massimo Santini

Author	Emiddio Ingenito @emikodes

Astrazione e decomposizione (Chapter 1)

All'aumentare della lunghezza (in numero di linee di codice) di un programma, la struttura monolitica (singolo file/procedura) diventa sempre meno ragionevole.

Il più semplice paradigma della programmazione, prevede la soluzione di problemi di grande entità, secondo tecniche di **decomposizione** e **astrazione**.

Decomposizione

Scomporre il programma principale, in unità indipendenti (moduli) di lunghezza minore

Questo permette di garantire che più persone possano occuparsi dell'implementazione del codice, indipendentemente dal lavoro degli altri.

Le soluzioni dei "sotto-problemi", possono essere combinate per risolvere il problema originale.

Astrazione

Visione alternativa di un problema: Cambiando il livello di dettaglio da dover considerare, posso “ignorarne” alcuni, riconducendo il problema ad una versione più semplice da analizzare.

In particolare, l’uso delle procedure nei linguaggi di programmazione, utilizza due metodi di astrazione possibili:

Abstraction by parameterization

Consideriamo per esempio una procedura “sort” che esegua l’ordinamento di un insieme di interi a:

E’ molto probabile, che durante il ciclo d’esecuzione del programma, noi vorremmo ordinare anche array diversi da a, sia per nome che per contenuto.

Generializziamo il comportamento della procedura “sort”, introducendo i parametri formali

> A sort routine that works on **any** array of integers is much more generally useful than one that works only on a particular array of integers.
>
> By further abstraction, we can achieve even more generality.
>
> For example, we might define a **sort** abstraction that works on arrays of floats as well as arrays of integers, or even one that works on array-like structures in general

Abstraction by specification

I Moduli implementati, possono essere utilizzati dai programmatori come fossero “black-box”, con la sola conoscenza degli effetti finali che il modulo realizza.

Realizziamo ciò associando alla procedura una “specifica”, un commento che sia sufficientemente informativo per permettere agli utenti di utilizzarla senza dover guardare il codice.

Un buon modo per scrivere una specifica, è associare al codice due informazioni:

- Precondizioni, devono essere assunte vere all’entrata della procedura
- Postcondizioni, devono essere assunte al termine dell’invocazione della procedura.

Esempio:

```
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));
        i = i + 1;
    }
    return ans;
}
```

Dalla specifica di questa procedura, capiamo due cose:

- Al termine della procedura, possiamo assumere che venga restituita “un’approssimazinone della radice quadrata del parametro passato”.
- Possiamo assumere ciò, solo se il parametro passato è >0.

Tipi di astrazione

Combinando i metodi di astrazione per parametrizzazione e specifica, possiamo definire tre tipi di astrazione:

- Astrazione procedurale - permette di aggiungere operazioni al codice
- Astrazione dei dati - permette di rappresentare nuovi tipi di oggetti
- Astrazione dell'iterazione - permette di iterare gli elementi di una collezione, senza rivelare come essi sono implementati.

Oggetti in Java (Chapter 2)

Java è un linguaggio object-oriented, ciò significa che la maggior parte dai dati, sono contenuti in oggetti, che memorizzano valori, e definiscono operazioni su di essi (metodi).

I Programmi Java, sono composti da classi e interfacce.

Le classi, possono essere utilizzate per definire collezioni di procedure (metodi)

```
public class Num {
   // class providing useful numeric routines
   public static int gcd (int n, int d) {
   // requires: n and d to be greater than zero
   // the gcd is computed by repeated subtraction
      while (n != d)
         if (n > d) n = n - d; else d = d - n;
         return n;
         }
   public static boolean isPrime(int p) {
   // implementation goes here
   }
}
```

O per implementare nuovi tipi di dati (anche in questo caso, avrò metodi che definiscono le operazioni che posso effettuare su tali tipi di dati)

Esempio: Classe **"Linked-List"**, contiene le procedure per manipolare una Lista.

Al suo interno, ci sono i metodi **isEmpty()** , **push()**, **pop()**...

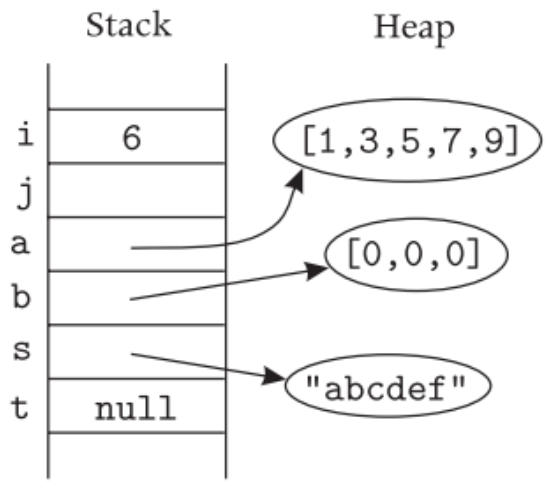
Packages

Classi e interfacce, sono raggruppate in “pacchetti”.

I Pacchetti, implementano un meccanismo di incapsulamento, infatti in base a regole di visibilità, le classi e le interfacce in un pacchetto potranno essere o meno utilizzate al di fuori di esso.

Variabili

Le variabili primitive, sono allocate nello stack, tutte le altre, incluse stringhe e array, sono memorizzate nello stack come riferimenti (puntatori) a oggetti allocati nello heap.




```
int i = 6;
int j; // errore
int [] a = {1,3,5,7,9}; // creates a 5-element array
int [] b = new int[3]; //array da 3 elementi (inizializzati a 0)
String s = "abcdef"; // creates a new string
String t = null;
```

Ogni variabile in Java, deve essere inizializzata al momento della dichiarazione

Gli oggetti allocati nello heap, sono creati con l’utilizzo dell’operatore **new**

```
int [ ] a = new int[3];
//array di 3 interi allocato nello heap,
```

```
//con un riferimento a tale oggetto, memorizzato in 'a'
```

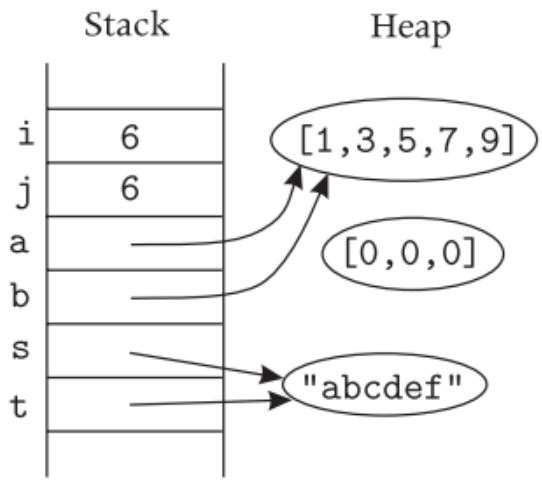
L'operatore new, rappresenta il metodo standard per creare un nuovo oggetto.

Crea un nuovo oggetto della classe indicata, e richiama il costruttore associato alla classe.

Per esempio, il costruttore della classe array, prevede l'inizializzazione di ogni elemento a 0.

Ogni oggetto creato con l'operatore new o con una forma "speciale" come per esempio `String str = "abcd"`, ha un'identità distinta da ogni altro oggetto del programma (Risiede in uno spazio a se sullo heap)

Gli assegnamenti tra variabili, copiano i valori, quelli tra oggetti copiano i riferimenti ad essi, esempio:



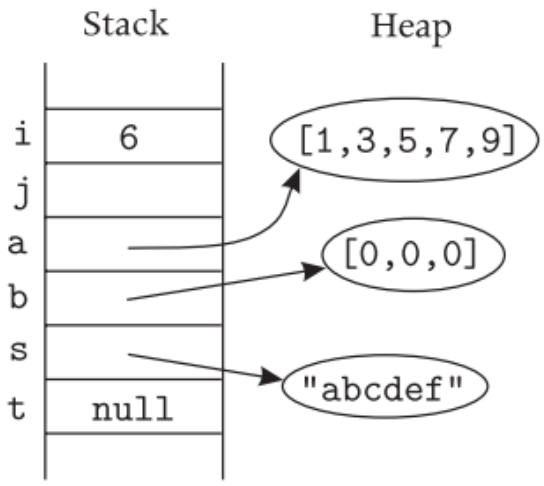

```
//copia il valore di i
//nella variabile j
j = i;

//copia dei riferimenti,
//gli stessi oggetti saranno
//condivisi tra due variabili
b = a;
t = s;
```

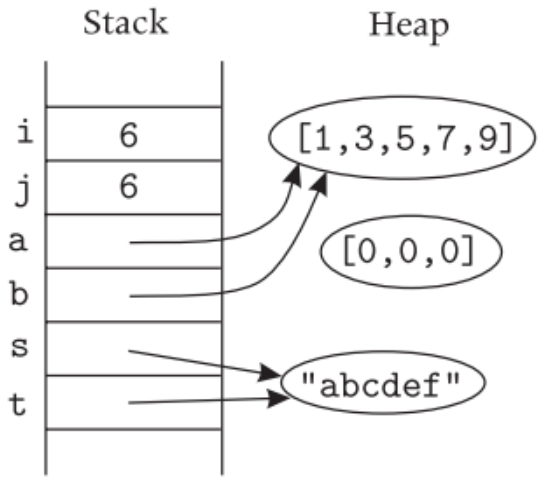
L'operatore `==`, permette di determinare se due variabili contengono **lo stesso valore.**

Nel caso di dati primitivi, questa comparazione è elementare, nel caso degli oggetti, vengono comparati i **valori delle variabili**, quindi gli **indirizzi di memoria**

sullo heap: in caso di comparazione di oggetti tramite l'operatore `==`, verrà restituito true solo se essi si riferiscono allo stesso oggetto.



Qui a==b è false



a==b è true

Gli oggetti vivono all'interno del programma, finchè esistono riferimenti ad essi: se un oggetto non è più raggiungibile, sarà prima o poi distrutto dal garbage collector.

Gli oggetti possono essere mutabili o meno, cioè il loro valore può cambiare o meno.

Esempio: Gli array sono mutabili, le stringhe sono immutabili.

Passaggio parametri

Quando un metodo viene chiamato `classe.metodo(param1, param2)`, viene valutata la classe, ed identificato il metodo da chiamare.

Vengono valutati i parametri della chiamata, da sinistra a destra, per estrarne i valori, poi viene creato un record d'attivazione sullo stack, che conterrà i parametri formali dell'header del metodo `public static metodo ( parametro formale1, parametro formale2 ){ }`, all'interno dei quali vengono copiati i valori dei parametri passati dal chiamante.

Vengono poi allocate le variabili locali, ed eseguito il corpo del metodo.

Il passaggio parametri avviene sempre per copia: se un parametro è un riferimento ad un oggetto, verrà passato l'indirizzo di memoria dello heap, pertanto la procedura chiamata condividerà gli oggetti col chiamante.

Se gli oggetti sono mutabili, e la procedura ne modifica lo stato, tali modifiche saranno visibili anche al chiamante.

Type Checking

Java è uno "**strongly typed language**", ciò significa che tutti gli errori relativi ai tipi, sono identificati a compile time.

Grazie a ciò, Java garantisce che i programmi che passano la fare di compilazione, non contengano errori relativi ad uso improprio dei tipi.

> For example, consider
> `int y = 7;`
> `int z = 3;`
> `int x = Num.gcd (z, y);`
> When the compiler processes the call to Num.gcd, it knows that Num.gcd
> requires two integer arguments, and it also knows that expressions z and y
> are both of type int.
> Therefore, it knows the call of gcd is legal. Furthermore, it knows that gcd returns an int, and therefore it knows that the assignment to x is legal.

Type Hierarchy

In Java, i tipi sono organizzati in una gerarchia di supertipi e sottotipi.

La relazione di sottotipo è transitiva (Se R è sottotipo di S, e S è sottotipo di T, allora R è sottotipo di T), e riflessiva (S è sottotipo di se stesso).

Se S è un sottotipo di T, allora posso usare S in tutti i contesti in cui mi aspetterei un tipo T.

Il compilatore Java, impone che S sottotipo di T, implementi tutti i metodi di T.

A capo della gerarchia dei tipi, c'è il tipo `Object`: tutti gli oggetti (String, Array...), sono sottotipo di `Object`.

Object implementa alcuni metodi come `equals` e `toString`, così come anche tutti i suoi sottotipi.

L'assegnamento `v=e` è valido solo se `e` è sottotipo di `v`

```
Object o1 = array; //valid
Object o2 = stringa; //valid

String s = "aaa" //valid, "aaa" è una stringa, String è sottotipo di String (Rifless iva)
```

Il tipo di una variabile, è ricavato a compile-time, tramite un controllo detto "di tipo apparente".

Il tipo apparente di o1 è `object`, il tipo reale sarà `array`

Il compilatore effettua type checking quindi solo in base ai tipi apparenti, quindi un utilizzo del tipo:

```
o2.length()
```

Sarà considerato illegale (il compilatore sa che il tipo di o2 è `Object` (tipo apparente): `Object` non ha il metodo `length`)

Il seguente utilizzo è valido:

```
o2.equals("abc")
```

Anche `String s = o2` è illegale (Stesso motivo)

Posso “raggirare” questo controllo, effettuando casting della variabile, prima di utilizzarne i metodi

```
(String)o2.length() //valid
```

Ora il controllo, è spostato a run-time. Verrà effettuato il casting di o2 a string, poi richiamato il metodo length()

> Fare molta attenzione nell’effettuare casting, se o2 non potrà essere convertito in String, il programma terminerà sollevando un’eccezione.

Overloading

Java permette di implementare più metodi, con lo stesso nome, ma diversi parametri.

Esempio: l’operatore +, è definito per operazioni tra interi, float, double, tra interi e double (...)

Ipotizzando di avere i seguenti metodi, con dichiarazioni diverse

```
static int comp(int, long) // defn. 1
static float comp(long, int) // defn. 2
static int comp(long, long) // defn. 3
```

Allora le chiamate saranno effettuate scegliendo il metodo “most specific”:

```
int x;
long y;
float z;

comp(x,y) //legale, richiama defn.1
comp(x,z) //illegale, non esiste comp(int,float)), errore a compile-time.
comp(y,y) //legale, richiama defn.3
comp(x,x) //illgale, non esiste comp(int,int), errore a compile-time.
comp((long)x,x) //legale, richiama defn.2
```

Dispatching

Quando richiamo un metodo su un oggetto, è necessario che venga richiamato il metodo corretto per quello specifico tipo.

Esempio:

```
String t = "ab";
Objecto=t+ "c"; // concatenation
String r = "abc";
boolean b = o.equals(r);
```

Quale metodo equals bisogna richiamare? quello di Object o quello di String?

Se viene richiamato quello di `Object`, verrà restituito `false`, se viene invece richiamato il metodo equals di `String`, che è stato sovrascritto per effettuare una comparazione lessicografica, verrà restituito `true`.

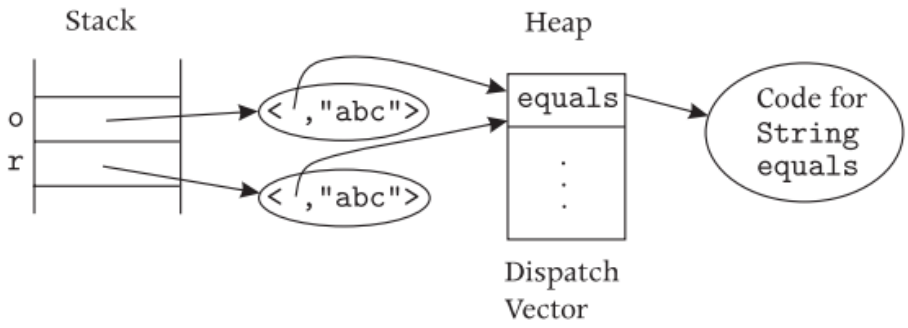
Se usassimo il tipo apparente per determinare quale metodo eseguire, o è un oggetto, quindi verrebbe eseguito il metodo di `Object`.

Introduciamo il meccanismo di `dispatching` di un metodo, per far si che venga richiamato il codice del tipo reale, e non apparente.

Il compilatore, genera per ogni classe una tabella di dispatching (vtable), che contiene i puntatori ai codici dei metodi specifici della classe.

Se una classe sovrascrive un metodo del suo supertipo, la voce corrispondente della vtable viene aggiornata con un puntatore al codice del metodo sovrascritto.

Quando un oggetto viene istanziato, viene associato alla vtable della sua classe reale: l'accesso ai metodi, viene effettuato a run-time consultando la vtable.



Types

Ci sono alcuni contesti, in cui solo i valori degli oggetti sono validi per l'utilizzo di alcuni metodi.

Possiamo utilizzare i tipi primitivi, come int e char, utilizzando le loro **"wrapper classes".**

Ogni tipo primitivo ha una classe associata (**Esempio**: **Integer** per **int**, **Character** per **char**)

Queste classi, "**wrappano**" i tipi di dati primitivi, in una struttura del tipo:

```
public final class Integer{

  private final int value;
```

```
    public Integer(int value){ //costruttore
        this.value = value;
    }

    public int intValue(){
        return value;
    }

}
```

Nella realtà nella classe ci sono molti altri metodi.

Esempio:

```
int n = Integer.parseInt("1024");
//n conterrà l'intero 1024, se la stringa non è un intero,
//viene generata un'eccezione NumberFormatException
```

Vectors

Un “vector” è una struttura indicizzata, un insieme di **oggetti**, di dimensione dinamica (si ridimensiona a run-time).

```
Vector<E> v = new Vector<E>(); //inizializzazione vector.

Vector<Integer> v = new Vector<Integer>(3);
//vector di Integer, di capacità 3

Vector<Integer> v = new Vector<Integer>(3,5);
```

```
//vector di Integer, di capacità 3
//quando riempio il vector, la struttura cresce di 5. (2nd param).
```

'E', è un oggetto, ma potrei anche inizializzare un **vector** per istanziare una collezione di dati eterogenea (in cui posso mettere oggetti di tipo diverso)

```
Vector v = new Vector(); // creates a new, empty Vector
```

Metodi principali:

- . **`size()`**
- . **`add(element)`**
- . **`add(index,element)`**
- . **`addAll(vector)`** - adds all elements of a vector to another vector
- . **`get(index)`** - returns an element specified by the index
- . `remove(index)` - removes an element from specified position
- . `clear()` - removes all elements.
- . `contains()` - searches the vector for specified element and returns a boolean result
- `v.set(0, "def")` - element at 0 position is set to "def".

Interfaccia "List"

L'interfaccia List, in Java, è implementata dalle classi ArrayList, LinkedList,Vector,Stack.

ArrayList è "costruito" utilizzando un array, LinkedList utilizzando una lista doppiamente connessa.

Entrambi non possono essere utilizzati con tipi di dato primitivi, dobbiamo utilizzare la “wrapper classes”.

ArrayList

```
ArrayList<Integer> obj = new ArrayList<>();
ArrayList<String> obj = new ArrayList<>();
```

Metodi dell’interfaccia List (comuni quindi, anche ai vector)

- `add(index, element)` : inserts the given object at the `index`, adjusting the index if there are remaining elements
- `get(index)` : returns the object at the given `index`
- `set(index, element)` : replaces the element at the given index with the new element
- `remove(index)` : removes the element at the given `index`, adjusting the index of the remaining elements.

The methods `indexOf(element)` and `lastIndexOf(element)` return the index of the given element in the list, or -1 if the element is not found.

The `subList(start, end)` returns a list consisting of the elements between indexes `start` and `end - 1`. If the indexes are invalid then an `IndexOutOfBoundsException` exception will be thrown.

Stream di input/output

L’I/O in Java, avviene utilizzando oggetti del supertipo Reader (o sottotipo), e Writer (o sottotipi)

Per esempio: BufferedReader è un sottotipo del tipo Reader, che permette di effettuare lettura bufferizzata (più caratteri contemporaneamente).

FileReader, è un sottotipo di Reader che permette la lettura del contenuto di un file.

Per semplificare le cose, Java mette a disposizione tre oggetti, per effettuare operazioni di lettura e scrittura da standard input, standard output e standard error.

```
System.in // Standard input to the program.
System.out // Standard output from the program.
System.err // Error output from the program.
```

“I Ferri del mestiere”

Input/output

Comand Line Arguments

> Per *argomenti sulla linea di comando* si intendono tutte le parole (stringhe massimali non contenenti spazio) che seguono il nome della classe nell’invocazione della JVM. Ad esempio, se avete compilato una classe di nome `Soluzione` e ne invocate l’esecuzione tramite l’interprete come
>
> `java Soluzione uno 2 tr_e`
>
> gli argomenti saranno le tre parole: `uno`, `2` e `tr_e`.
>
> ```
> return_type pname (...)
> // requires: This clause states any constraints on use
> // modifies: This clause identifies all modified inputs
> // effects: This clause defines the behavior
> ```

Gli argomenti, sono accessibili dalla funzione “main”, nell’array “args”.

Notiamo che “args” è un array di stringhe, quindi dovrei poter utilizzare i metodi di parsing delle wrapper classes, in caso ricevessi argomenti numerici.

```
public class SommaArgs {
  public static void main(String[] args) {
    int somma = 0;
    for (String arg : args)
      somma += Integer.parseInt(arg);
    System.out.println(somma);
  }
}
```

Lettura input con blocco try-with-resources

Il Blocco try-with-resources, permette di gestire in automatico errori derivanti da errata lettura (exception handler), e di allocare una risorsa, che verrà rilasciata alla fine del blocco.

In generale, il codice avrà la seguente struttura:

```
try (/* Oggetto utilizzato per ricavare input */){
 while (/* c'è input */)
   /* consuma l'input */
}
```

L’Oggetto utilizzato per l’input, varia in base a se voglio leggere “riga per riga” (1.1), lettura tokenizzata (1.2), e se voglio leggere da flusso standard (2.1) o da file (2.2)

In base alla funzione da realizzare, gli oggetti saranno istanziati come:

```
(1.1., 2.1.)  BufferedReader in = new BufferedReader(new InputStreamReader(S
ystem.in));
```

```
(1.1., 2.2.)  BufferedReader in = new BufferedReader(new FileReader(path));
(1.2., 2.1.)  Scanner in = new Scanner(System.in);
(1.2., 2.2.)  Scanner in = new Scanner(new FileInputStream(path));
```

Per consumare gli input:

```
//lettura riga per riga
String linea = null;
while ((linea = in.readLine()) != null)
  /* consuma l'input */


 //lettura interi
 while (in.hasNextInt()) {
  int intero = in.nextInt();
  /* consuma l'input */
}

//lettura stringhe (whitespace)
while (in.hasNext()) {
  String stringa = in.next();
  /* consuma l'input */
}
```

Astrazione procedurale (Chapter 3)

L’astrazione procedurale, combina i metodi di astrazione parametrica e specifica (Chapter 1).

L’Astrazione parametrica, è ottenuta associando una parte semantica al codice della procedura, che spieghi “cosa la funzione fa”.

Benefits:

- Locality (località): basta leggere la documentazione per capire cosa fa il codice
- Modifiability (modificabilità): Posso cambiare l'implementazione delle procedure ma, a patto che le specifiche non cambino, cioè che le clausole requires, modifies, ed effects rimangano valide, gli utilizzatori non ne risentiranno.

Template per la scrittura della specifica di una procedura:

La clausola requires, include tutte le condizioni da soddisfare, per utilizzare correttamente la procedura.

Questa clausola è necessaria, se ci sono valori di parametri per cui il comportamento della procedura non è definito (in tal caso la procedura è detta parziale.)

Esempio: la funzione va in loop se viene fornito un intero dispari, la clausola requires allora indicherà che il parametro deve essere pari

La clausola modifies, include tutti gli input, espliciti (parametri) o impliciti (files) modificati dalla funzione.

La clausola effects, descrive gli effetti della procedura, gli output, e come gli input specificati nella clausola modifies sono modificati.

In Java, procedure "standalone" sono scritte come metodi statici di una classe.

Anche la classe stessa, ha una piccola parte semantica

```
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

    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;
        }
     }

  }
```

> Nota come i metodi quickSort e partition, sono private, e il loro uso è limitato all'interno della classe Arrays. Ciò ha senso, perchè sono solo "routines di appoggio" per la procedura sort, che invece è utilizzabile al di fuori della classe.

Proprietà dell'astrazione procedurale:

- Minimalità - Una procedura è più minimale di un'altra, se prevede meno condizioni per operare.
- Comportamento sottodeterminato - se per un input, la procedura può fornire più di un output
- Comportamento deterministico - dato un input, l'output prodotto è sempre uguale
- Generalità - Una procedura è più generale di un'altra se può gestire una classe più ampia di input (Esempio: una procedura che lavora su uno specifico array di interi di dimensione 5. è meno generale di una che lavora con qualsiasi array di interi di dimensione 5, che a sua volta è meno generale di una che lavora su array di interi di qualsiasi dimensione)

Commenti Javadoc

In Java esistono tre tipi di commenti, inline, multi-line, e commenti Javadoc.

I commenti Javadoc permettono di generare una documentazione HTML del nostro codice, tramite un parser di commenti (Javadoc)

Doc Comment

Precede una classe, costruttore o metodo.

E' Composto da due parti, una descrizione, e una serie di tags.

```
/** apertura commento Javadoc
 short description goes here
 <p>
 long description goes here

 @author Emiddio Ingenito solo per classi e interfacce
 @version
 param e return, solo per metodi e costruttori
 @param  url  an absolute URL giving the base location of the image
 @param  name the location of the image, relative to the url argument
 @return      the image at the specified URL

 @throws Exception
 @see Image


*/ chiusura commento Javadoc
public Image getImage(URL url, String name) {
    try {
    return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
```

```
    return null;
    }
}
```

Exceptions (Chapter 4)

L'utilizzo di procedure parziali, il cui comportamento non è definito per alcuni valori di input, porta alla scrittura di programmi poco robusti.

La robustezza di un programma, indica la capacità di esso di comportarsi "ragionevolmente" anche in presenza di errori.

Posso aumentare la robustezza, modificando le procedure parziali, in procedure totali, introducendo dei controlli che informino il chiamante in caso la procedura non potesse operare correttamente.

Esempio:

```
public static int fact (int n)
// effects: If n > 0 returns n! else returns 0.
```

Ho stabilito una sorta di "codice di errore". Il chiamante controllerà il valore di ritorno di "fact" e in caso fosse pari a 0, saprà che la chiamata alla procedura è stata effettuata in modo "illegale".

```
int r = Num.fact(y);
if (r > 0) z = x + r; else ...
```

Questo approccio del "restituire un valore specifico come codice di errore", non è sempre applicabile.

Se il "codominio" della funzione non prevedesse valori inutilizzati? E' meglio trattare gli errori tramite il meccanismo delle eccezioni.

Applicando il meccanismo delle eccezioni, una procedura può terminare o restituendo un valore, o lanciando un’eccezione.

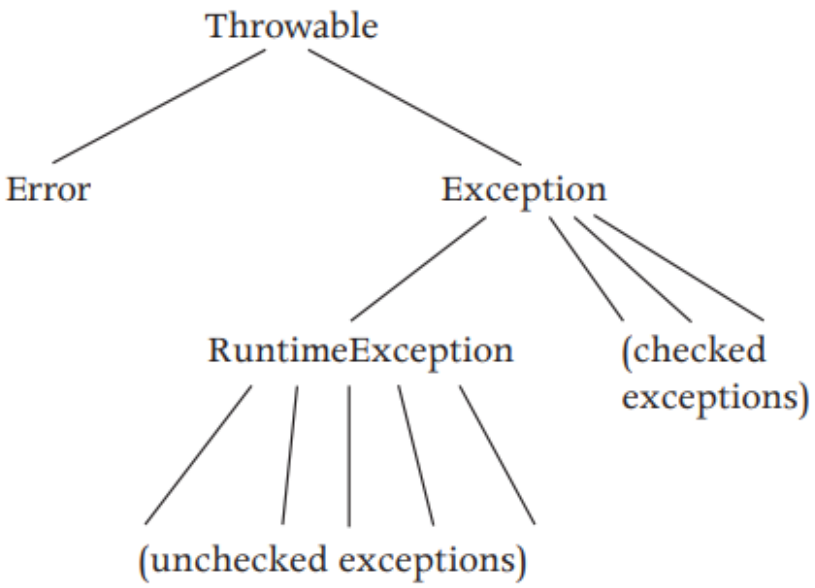
Utilizzo delle eccezioni

Indichiamo che tipo di eccezioni la procedura può lanciare, direttamente nel suo header.

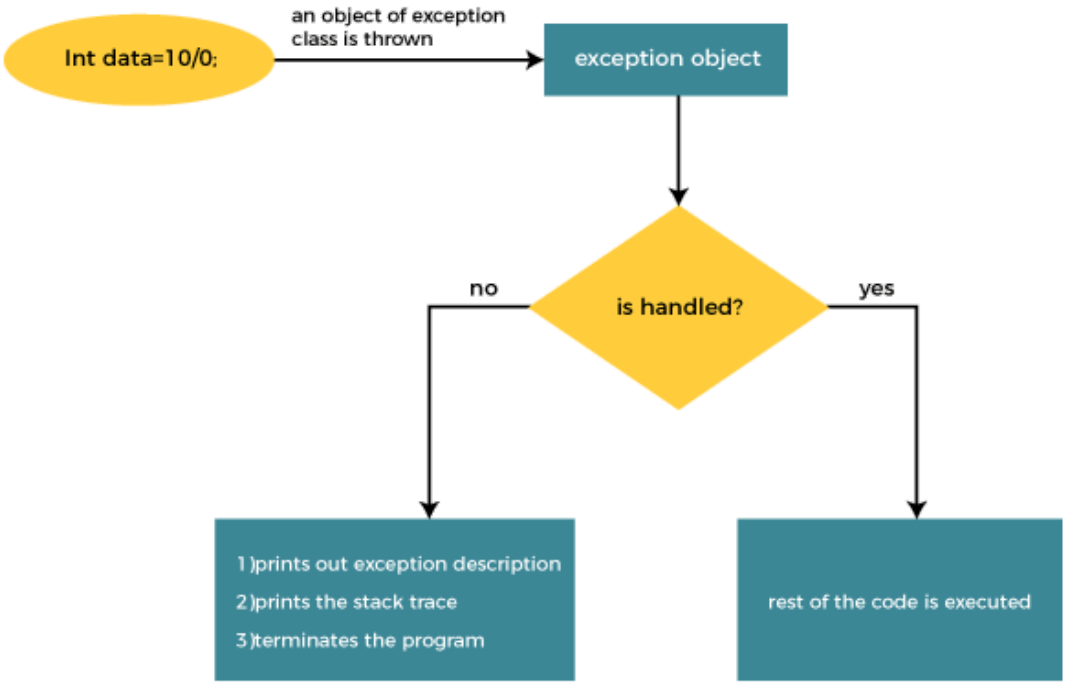
```
public static int fact (int n) throws NonPositiveException
// effects: If n is non-positive, throws NonPositiveException, else
// returns the factorial of n

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

Tipi di eccezioni



Tutte le eccezioni, sono sottotipi del tipo “Throwable”, e possono essere checked, o unchecked.



- Checked - Devono essere previste, e gestite in modo esplicito (come mostrato di seguito, con try-catch)

Vengono usate per gestire errori che dipendono da risorse esterne, come file mancanti, problemi di rete, o database non raggiungibili. Questi errori si verificano per cause che non dipendono dal codice in sé, quindi l'obiettivo è prevedere e gestire questi errori in modo elegante.

> Le eccezioni checked, devono essere singolarmente documentate mediante l'utilizzo del tag Javadoc "@throws"

- Unchecked - Indicano errori di logica o di programmazione, sono quelle più gravi (Esempio: accesso a indice non valido di un array).

  Vanno risolte direttamente correggendo il codice, pertanto possono non essere gestite nel codice chiamante.

Tabella riassuntiva delle eccezioni più utilizzate

Exception	Occasion for Use
`IllegalArgumentException`	Non-null parameter value is inappropriate
`IllegalStateException`	Object state is inappropriate for method invocation
`NullPointerException`	Parameter value is null where prohibited
`IndexOutOfBoundsException`	Index parameter value is out of range
`ConcurrentModificationException`	Concurrent modification of an object has been detected where it is prohibited
`UnsupportedOperationException`	Object does not support method

Definire nuovi tipi di eccezioni

Se l’eccezione estende il supertipo “Exception”, è di tipo checked, se estende “RuntimeException”, è unchecked.

```
public class NewKindOfException extends Exception {
   public NewKindOfException( ) { super( ); }
   public NewKindOfException(String s) { super(s); }
}
```

Il costruttore dell’eccezione, è “sovraccaricato” (overloaded), per gestire anche l’inizializzazione di una stringa “s”, che indica il perchè l’eccezione è stata sollevata

```
Exception e1 = new NewKindOfException("this is the reason");
Exception e2 = new NewKindOfException( );
```

La spiegazione del perchè l’eccezione è stata sollevata, può essere ottenuta richiamanto il metodo “.toString()”

```
String s = e1.toString( );
//s → "NewKindOfException: this is the reason"
```

Lanciare eccezioni

Terminiamo una procedura con un’eccezione, con il metodo throw

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

Gestione eccezioni

Utilizzo del costrutto “try-catch”

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

Per gestire più eccezioni

```
try {
  //.....
} catch ( IllegalArgumentException | SecurityException |
        IllegalAccessException| NoSuchFieldException exc) {
  someCode();
}
```

Per gestire eccezioni diverse

```
try {
    ...
} catch (IllegalArgumentException e) {
    someCode();
} catch (SecurityException e) {
    someCode();
} catch (IllegalAccessException e) {
    someCode();
} catch (NoSuchFieldException e) {
    someCode();
}
```

Non abusare del costrutto try-catch quando non necessario. Questo perchè il codice all'interno del try-catch, non viene ottimizzato al 100% dalla JVM, rendendolo due volte più lento rispetto a test con condizioni esplicite!

"Exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow."

Inoltre, favorire l'uso di eccezioni standard, senza crearne di nuove, per rendere il codice prestazionalmente migliore, oltre che più leggibile (i programmatori conoscono già le eccezioni standard, senza dover leggere la documentazioni delle eccezioni specifiche del nostro codice)

Data abstraction (Chapter 5)

Permette di estendere il linguaggio di programmazione, introducendo nuovi tipi di dati (istanze di classi, oggetti).

L'astrazione dati, consiste in insiemi di oggetti, e di operazioni effettuabili su di essi.

Per esempio, in un contesto matematico, possiamo introdurre il tipo "polinomio", ed implementare le operazioni di somma, moltiplicazione, sottrazione tra essi, oppure un tipo "matrice", con le relative operazioni.

L'astrazione dati, utilizza astrazione per parametrizzazione (uso di parametri nelle procedure), e astrazione per specificazione: questa è ottenuta includendo le operazioni all'interno del tipo di dato.

Facciamo un esempio: quando scegliamo di rappresentare un nuovo tipo, dobbiamo scegliere un metodo di memorizzazione delle sue informazioni (scegliere strutture dati adeguate).

Una volta scelta la rappresentazione, tutti i programmi dipenderanno da questa, per il corretto funzionamento.

Ma se nelle future versioni del codice, si decidesse di cambiare la rappresentazione? Esempio: scelgo di rappresentare un polinomio mediante una lista di interi, allora i programmatori scriveranno le proprie funzioni di somma, moltiplicazione ecc.. basandosi sul fatto che “un polinomio è una lista di interi”, ma se in un futuro aggiornamento, si decidesse di cambiare rappresentazione, con una linked list per esempio?

Tutti i programmi che utilizzano il tipo “polinomio”, devono essere modificati.

Se invece includiamo le operazioni all’interno del tipo, noi forziamo i programmatori ad utilizzare le operazioni specifiche, senza doverne implementare di proprie.

In altre parole, i programmatori non dovranno più scrivere le proprie operazioni somma, moltiplicazione ecc.. ma utilizzeranno i metodi messi a disposizione dalla classe.

Se si decidesse di cambiare rappresentazione, si dovranno riscrivere anche i metodi delle operazioni, ma i programmi che utilizzano la classe, non risentirebbero di tale riscrittura, poiché **non accedono direttamente alla rappresentazione.**

$$data\ abstraction = < objects, operations >$$

Ogni classe, prevede:

- Attributi (variabili)
- Metodi di creazione delle istanze/oggetti (costruttore, con lo stesso nome della classe, visibilità public)
- Metodi di mutazione, cambiano valore degli attributi
- Metodi di osservazione, ottengono valore degli attributi
- [Opzionale] Metodi di produzione, producono altri oggetti dello stesso tipo

Ogni oggetto fa riferimento allo spazio nello heap.

Per supportare l'astrazione, è bene che gli attributi siano dichiarati come "`privati`", e che il loro accesso avvenga esclusivamente mediante metodi osservazionali e di mutazione.

Le dichiarazioni di variabili non utilizzano la keyword "`static`", poiché appartengono agli oggetti (ogni oggetto, ha attributi diversi)

Negli oggetti mutabili: sì creatori, sì di osservazione, sì di modificazione, di produzione non è generalmente necessario.
Negli oggetti immutabili: sì creatori, sì di osservazione, no metodi modificazionali, sì di produzione

Gli oggetti mutabili sono generalmente più efficienti, mentre quelli immutabili sono più sicuri. Bisogna bilanciare le due cose e scegliere quello che più si confà all'entità che andiamo a rappresentare.

Esempio di specifica di una classe:

```
visibility class dname {
// overview: A brief description of the behavior of the type's objects goes her
e.
// constructors
// specs for constructors go here
// methods
// specs for methods go here
}
```

- Metodi e costruttori, appartengono più agli oggetti che alla classe stessa, pertanto con essi non compare la keyword "`static`"

  > Static, significa che i metodi definiti, non appartengono ad un oggetto specifico, ma alla classe, e possono essere pertanto utilizzati senza dover prima creare un oggetto.

- L’oggetto a cui appartiene un metodo o un costruttore, è riferibile implicitamente attraverso l’utilizzo della keyword “`this`”.

Esempio implementazione classe IntSet (Mutable):

```
public class IntSet {
   // overview: IntSets are mutable, unbounded sets of integers.
   // A typical IntSet is {x1, ... , xn}.
   // constructors
   public IntSet ( )
   // effects: Initializes this to be empty.
   // methods
   public void insert (int x)
   // modifies: this
   // effects: Adds x to the elements of this, i.e., this_post = this + { x }.
   public void remove (int x)
   // modifies: this
   // effects: Removes x from this, i.e., this_post = this-{x}
   public boolean isIn (int x)
   // effects: If x is in this returns true else returns false.
   public int size ( )
   // effects: Returns the cardinality of this.
   public int choose ( ) throws EmptyException
   // effects: If this is empty, throws EmptyException else
   // returns an arbitrary element of this.
}
```

Esempio implementazione classe Poly(Im-Mutable):

Una volta creato e istanziato utilizzando un costruttore, l’oggetto Poly non può essere modificato.

```
public class Poly {
   // overview: Polys are immutable polynomials with integer coefficients.
   // A typical Poly is c0 + c1x + ...
```

```
    // constructors
    public Poly ( )
    // effects: Initializes this to be the zero polynomial.
    public Poly (int c, int n) throws NegativeExponentException
    // effects: If n<0 throws NegativeExponentException else
    // initializes this to be the Poly cxn.
    // methods
    public int degree ( )
    // effects: Returns the degree of this, i.e., the largest exponent
    // with a non-zero coefficient. Returns 0 if this is the zero Poly.
    public int coeff (int d)
    // effects: Returns the coefficient of the term of this whose exponent is d.
    public Poly add (Poly q) throws NullPointerException
    // effects: If q is null throws NullPointerException else
    // returns the Poly this + q.
    public Poly mul (Poly q) throws NullPointerException
    // effects: If q is null throws NullPointerException else
    // returns the Poly this * q.
    public Poly sub (Poly q) throws NullPointerException
    // effects: If q is null throws NullPointerException else
    // returns the Poly this – q.
    public Poly minus ( )
    // effects: Returns the Poly – this.
}
```

Records

Per creare record (struttura dati composta da più parti), molti linguaggi forniscono funzionalità “built in” (Esempio: in C usiamo le struct)

In Java, essi sono implementati utilizzando classi.

```
public class Point {
    public int x;
    public int y;
```

```
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

E usare poi

```
public class main {
    public static void main(String[] args) {
            Point punto = new Point(2,3);
            System.out.println("il punto ha x pari a " + punto.x + "e y pari a " + pun
to.y);
    }
}
```

Per modellare dati immutabili, cioè il cui valore non deve essere modificato nel tempo, il modo migliore è implementare una classe immutabile.

Per garantire l'immutabilità, una classe deve rispettare i seguenti principi:

1. **Tutti i campi devono essere** `private` **e** `final` → In questo modo, il valore dei campi non può essere modificato dopo l'assegnazione nel costruttore.
2. **Non deve avere metodi setter o altri metodi che modificano lo stato interno.**
3. **Deve essere progettata in modo tale che non possa essere estesa** → Questo evita che una sottoclasse possa introdurre mutabilità. Si ottiene dichiarando la classe `final` oppure rendendo i costruttori `private` e usando metodi statici di fabbrica.
4. **Se contiene riferimenti a oggetti mutabili, questi non devono essere direttamente modificabili dall'esterno** → Si può fare una copia difensiva degli oggetti mutabili in ingresso e in uscita.

```
public class Point {
    private final int x;
```

```
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX(){
        return this.x
    }

    public int getY(){
        return this.y
     }
}
```

Le variabili sono private e final (come const), e avendo solo metodi “getter”, il loro valore è read-only.

Al posto di scrivere classi immutabili a mano, Java mette a disposizione la keyword “record”, che crea automaticamente classi immutabili per noi,

```
public record Point(int x, int y) {}
```

Verrà creata una classe immutabile equivalente, con le variabili (final) x e y, i suoi getter, e altri metodi, come scritto qui sotto

```
import java.util.Objects;

public final class Point {
    private final int x;
    private final int y;

    // Costruttore compatto che inizializza i campi
    public Point(int x, int y) {
```

```
        this.x = x;
        this.y = y;
    }

    // Metodi getter per le proprietà x e y
    public int x() {
        return x;
    }

    public int y() {
        return y;
    }

    // Override di equals() per confronto basato sui valori
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    // Override di hashCode() per generare un hash basato sui valori
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    // Override di toString() per una rappresentazione leggibile
    @Override
    public String toString() {
        return "Point[x=" + x + ", y=" + y + "]";
    }
}
```

Quando scriviamo un record, possiamo anche ridefinire il costruttore, per esempio per definire alcune regole sulla memorizzazione dei dati.

```
public record Range(int start, int end) {

    public Range(int start, int end) {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser than star
t");
        }
        if (start < 0) {
            this.start = 0;
        } else {
            this.start = start;
        }
        if (end > 100) {
            this.end = 10;
        } else {
            this.end = end;
        }
    }
}
```

Qui forziamo che la fine sia maggiore o uguale dell'inizio, start non può essere negativo, end non può essere > 100.

Altri metodi

Ogni oggetto, è un sottotipo del tipo "Object", e pertanto ne eredita tutti i metodi.

Alcuni metodi definiti da "Object" sono:

- **obj.Equals(secondObj)** → Restituisce true, se l'oggetto passato come parametro, è uguale ad obj.

  Di default, il metodo equals per la classe Object implementa la più discriminante relazione di equivalenza possibile: per ogni x,y riferimenti non

nulli, restituisce true solo se x e y sono riferimenti allo stesso oggetto.

Se questa definizione non si applica per il nostro caso, equals va sempre sovrascritto, nel caso la classe fosse utilizzata per memorizzare dei valori (@override), e deve essere implementato in moduo da aderire al contratto di “relazione di equivalenza”

> The equals method implements an equivalence relation. It has these properties:
> • Reflexive: For any non-null reference value x, x.equals(x) must return true.
> • Symmetric: For any non-null reference values x and y, x.equals(y) must return true if and only if y.equals(x) returns true.
> • Transitive: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
> • Consistent: For any non-null reference values x and y, multiple invocations of x.equals(y) must consistently return true or consistently return false, provided no information used in equals comparisons is modified.
> • For any non-null reference value x, x.equals(null) must return false

Esempio di implementazione metodo equals, su classe IntSet:

```
@Override
public boolean equals(Object obj) {
 if (obj == null) return false;
if (obj istanceof IntSet) {
/*controllare se l'oggetto è
```

```
effettivamente un IntSet per evitare errori
unchecked non presenti nel
contratto*/
IntSet other = (IntSet)obj; // si converte obj in un IntSet
if (els.size() != obj.els.size()) return false;
// ..
return true;
 } else {
return false;
 }
}
public int hashCode() {
 // manca hashCode
}
```

Il Metodo equals è molto difficile da implementare.

Se ci troviamo in una di queste circostanze, possiamo evitare di sovrascrivere equals, e utilizzare quello di default:

- Ogni istanza della classe è già univoca (Esempio: Threads, ognuno ha già un proprio attributo PID)
- Non c'è utilità nel sovrascrivere equals
- equals è già stato sovrascritto dalla superclasse, e tale comportamento si applica anche al sottotipo.
- La classe è privata, e siamo quindi sicuri che equals non sia mai chiamato accidentalmente.

  Per essere sicuri che non sia mai chiamato, possiamo anche sovrascrivere per generare un'eccezione

  ```
  @Override
  public boolean equals(Object o) {
     throw new AssertionError(); // Method is never called
  }
  ```

> For mutable types, objects are equals only if they are the very same object, while for
> immutable types, they are equals if they have the same state.

- **obj.clone()** → Restituisce un oggetto con lo stesso stato di esso.

  Non utilizzato, al suo posto useremo i “copy constructors”, un costruttore che prende in ingresso un oggetto dello stesso tipo, e ne produce una copia che sia il più simile possibile (copiandone gli attributi)

  Esempio:

```
class MyClass {
    int value;
    // Default constructor
    public MyClass(int value) {
        value = value;
     }
    // Copy constructor
    public MyClass(MyClass other) {
        value = other.value;
     }
    public void display() {
        System.out.println("Value: " + value);
     }
    }

    public class Main {
        public static void main(String[] args) {
            MyClass original = new MyClass(10);
            MyClass copy = new MyClass(original); // Using copy constructor
            original.display(); // Output: Value: 10
            copy.display(); // Output: Value: 10
```

```
    }
}
```

```
public IntSet(IntSet other) {
   els = new ArrayList<>(other.els);
}
```

- **obj.toString()** → Restituisce una stringa che mostra il tipo e lo stato attuale dell'oggetto.

  Ogni classe, deve sovrascrivere (**override**) **.toString().**

  ```
  @override
  ```

  La chiocciola indica un'annotazione Java, una sorta di metadato che esprime al compilatore come trattare il codice.

  In questo caso, chiariamo al compilatore e ai lettori del codice, che il metodo nella sottoclasse, non è un nuovo metodo con lo stesso nome (polimorfismo), ma vuole sovrascrivere il metodo .toString() della superclasse.

  Esempio metodo toString() per IntSet:

  ```
  @override
  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 + "}"; }
  ```

  Output: "IntSet: {1, 7, 3}"

- obj.hashCode → Quando sovrascriviamo .equals, dobbiamo sempre sovrascrivere anche hashCode

> Here is the contract, adapted from the Object specification :
> • When the hashCode method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value, provided no information used in equals comparisons is modified.
> This value need not remain consistent from one execution of an application to another.
>
> • If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result.
>
> • If two objects are unequal according to the equals(Object) method, it is not required that calling hashCode on each of the objects must produce distinct results. However, the programmer should be aware that producing distinct results for unequal objects may improve the performance of hash tables.

Il modo più semplice per sovrascrivere .hashCode, è utilizzare al suo interno il metodo statico hash

```
// One-line hashCode method - mediocre performance
@Override public int hashCode() {
 return Objects.hash(lineNum, prefix, areaCode);
}
```

StringBuilder

Una stringa, è una sequenza di caratteri immutabile.

Quando concateno più stringhe, in maniera sequenziale (in un loop), vengono create più copie intermedie della stringa, occupando molta memoria.

> When concatenating multiple strings in a sequential manner, especially inside a loop, we might encounter the problem of intermediate object overload in the heap.

Esempio:

```
int array[4]=new int[4]{1,2,3,4};
String s = "L'Array è";

for (int num : array){ //foreach
	s+=num
}

System.out.println(s)

//output: "L'Array è 1234"
```

Questo genera nello heap, una serie di “oggetti spazzatura”:

```
//Nello heap
L'Array è
L'Array è 1
L'Array è 12
L'Array è 123
L'Array è 1234
```

Ad ogni iterazione, una nuova stringa è allocata sullo heap.

Invece di usare il loop come prima, usiamo

```
StringBuilder builder = new StringBuilder();
for (String s : strings){
   builder.append(s)
}
String result = builder.toString()
```

In questo modo, non verranno create copie intermedie.

Abstraction function

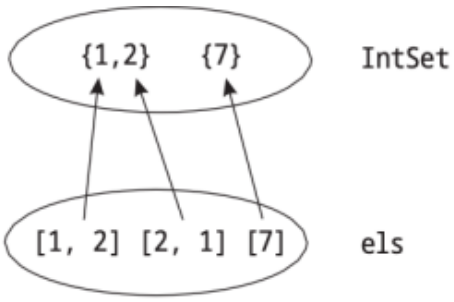
L'**Abstraction Function (AF)** descrive **come** lo stato interno (i campi di una classe) rappresenta un concetto astratto.

La ***Abstraction Function***, è una funzione $AF : C \rightarrow A$, che mappa le rappresentazioni degli oggetti concreti c della classe C, agli oggetti astratti $a \in A$ della vita reale che sto rappresentando.

$AF(c)$ rappresenta quindi un'oggetto della vita reale, espresso in base agli attributi della classe.

Esempio: IntSet

Ogni Insieme di Interi (oggetto astratto $a \in A$), è rappresentato nella classe da un vettore.



Questa funzione di astrazione, mappa il vettore els, contenuto all'interno della classe, ad un insieme di interi.

Mappatura di tipo molti-a-uno. I vettori [1,2] e [2,1], rappresentano lo stesso insieme di interi nella vita reale.

```
Funzione di astrazione di IntSet → AF(c) = {c.els[i].intValue | 0 <= i < c.els.siz
```

```
e}
```

Cioè, l'oggetto astratto a rappresentato dall'oggetto c, è composto da tutti gli interi all'interno del vettore `els`.

Esempio: Poly

Abbiamo scelto di rappresentare un polinomio, attraverso un vettore, in cui ogni elemento in posizione i, rappresenta il coefficiente di x^i.

Un polinomio a, è rappresentato tramite il vettore `terms` come:

$$A(c) = c_0 + c_1x + c_2x^2 \dots$$
$$dove\ c_i = c.terms[i]\ se\ 0 \leq i < c.terms.size,\ oppure\ c_i = 0\ altrimenti.$$

A livello di codice, oltre ad un commento nella classe, è implementato attraverso il metodo pubbico `toString`, dando per scontato che `String` sia isomorfo al mondo (cioè che l'oggetto astratto si possa rappresentare come una stringa)

Representation invariant

Potrebbe capitare, che non tutti gli oggetti della classe forniscano una legittima rappresentazione dell'oggetto astratto della vita reale.

Esempio: IntSet

Un IntSet è un oggetto astratto $a \in A$, che rappresenta un insieme di numeri interi, senza duplicati.

Avendo scelto un vettore per implementare la classe concreta $c \in C$ IntSet, questo potrebbe potenzialmente (il vettore) contenere elementi duplicati.

Introduciamo il ***Representation Invariant***, un predicato che tutte le legittime rappresentazioni concrete $c \in C$ devono rappresentare.

Esprimiamo questo concetto matematicamente:

```
// The rep invariant for IntSet is:
// c.els != null &&
// all elements of c.els are Integers &&
// there are no duplicates in c.els
```

Se tutti i possibili oggetti sono legittime rappresentazioni dell'oggetto astratto, la Rep. Invariant è semplicemente:

```
// The rep invariant is
// true
```

Va inserita come commento all'interno della classe, così che non compaia nella Javadoc.

```
public class IntSet{
   //Fields

   /*-
   * RI:
   *
   * c.els != null &&
   * all elements of c.els are Integers &&
   * there are no duplicates in c.els
   *
   *
   * AF(c) = {c.els[i].intValue | 0 <= i < c.els.size}
   */

   //Constructor
   ...
   //Methods
}
```

> Nel codice, la Liskov implementa la RI tramite il metodo `boolean repOk()`, che deve essere richiamato ogni volta che eseguo modifiche agli attributi dell'oggetto, per verificare che esso sia ancora una rappresentazione legittima dell'oggetto astratto.
>
> Piuttosto che richiamare un metodo che ogni volta controlli la legittimità di una classe, conviene non scrivere `repOk()`, ma effettuare controlli direttamente sui parametri aggiornati, per far si che la rep non venga violata, eventualmente sollevando eccezioni.

Nella scrittura del nostro codice, dobbiamo far si che la RI non venga mai violata (Never expose the rep).

Se una parte della rappresentazione (variabile) viene esposta all'esterno della classe, si perde il controllo sulla rep: l'utente potrebbe modificare tale variabile, invalidandola.

1. **Attributi non privati**

   Se gli attributi della classe non sono dichiarati `private`, l'utente può accedervi e modificarli direttamente, violando il controllo della classe.

2. **Restituire riferimenti alla rappresentazione interna (metodi osservazionali)**

   Anche con gli attributi dichiarati `private`, un metodo che restituisce direttamente una struttura dati mutabile interna (ad esempio un `Vector` o una `List`) espone la rappresentazione. L'utente, modificando quell'oggetto, modifica anche lo stato interno della classe.

   Esempio problematico:

```
public Vector allEls() {
   return els;  // els è un Vector privato che rappresenta i dati interni
}
```

Questo codice espone `els`,che essendo un riferimento, può essere modificato dall'utente, invalidando la rep.

Soluzione:

Restituire una copia della rappresentazione al posto del riferimento:

```
public Vector allEls() {
   return new Vector(els);  // Restituisce una copia di els
}
```

3. **Accettare oggetti mutabili forniti dall'esterno (costruttori o metodi mutazionali)**

   Se un costruttore o un metodo mutazionale accetta un oggetto mutabile fornito dall'esterno e lo salva direttamente nella rappresentazione interna, l'utente potrebbe passare parametri non validi, e invalidare la rep.

   Esempio problematico:

```
public IntSet(Vector elms) {
   if (elms == null) throw new NullPointerException();
   els = elms;  // Salva direttamente il riferimento al Vector fornito dall'utent
e
}
```

   Inoltre, l'utente può modificare `elms` dopo aver creato l'oggetto `IntSet`, rompendo il contratto della classe.

   Soluzione:

   Creare una copia dell'oggetto fornito:

```
public IntSet(Vector elms) {
   if (elms == null) throw new NullPointerException();
   els = new Vector(elms);  // Salva una copia di elms
```

```
}
```

Come evitare di esporre la rappresentazione?

1. **Attributi privati**

   Dichiarare tutti gli attributi della classe come `private`.

2. **Restituire copie di oggetti mutabili**

   Nei metodi osservazionali, restituire copie degli oggetti interni invece di riferimenti diretti.

3. **Creare copie degli oggetti mutabili forniti in input**

   Nei costruttori o nei metodi mutazionali, copiare gli oggetti mutabili forniti dall'utente.

4. **Usare oggetti immutabili**

   Dove possibile, usare oggetti immutabili per rappresentare i dati interni. Ad esempio, preferire `Collections.unmodifiableList` o `List.of` a una `List` mutabile.

5. **Fare adeguati controlli nei metodi mutazionali (costruttore o altri, bisogna assicurarsi che all'entrata e all'uscita di tali metodi, la RI sia vera)**

Non mostrare i dettagli implementativi

Mostrare i dettagli implementativi, anche attraverso metodi osservazionali apparentemente innocui, può vincolare l'evoluzione del codice.

Se l'API espone un'astrazione che riflette troppo da vicino la rappresentazione interna dei dati, si crea un legame stretto (tight coupling) tra l'implementazione e l'interfaccia pubblica della classe.

Questo legame limita la libertà dello sviluppatore nel cambiare l'implementazione.

Esempio del problema con un metodo osservazionale ingenuo

Immaginiamo una classe che rappresenta un insieme di dati (ad esempio, un insieme di numeri). In una prima implementazione, i dati sono memorizzati in una `List`. Lo sviluppatore offre il seguente metodo osservazionale:

```
public List<Integer> getElements() {
    return new ArrayList<>(elements);
    // Restituisce una copia della lista
}
```

Problemi con questa scelta:

1. **Vincolo sull'implementazione interna**

   Anche se il metodo restituisce una copia (quindi evita di esporre direttamente la rappresentazione interna), vincola lo sviluppatore a mantenere un'implementazione basata su una lista o, comunque, un'implementazione che possa essere facilmente trasformata in una lista.

   Supponiamo che in futuro lo sviluppatore decida di:

   - Salvare i dati in un database.
   - Usare una struttura dati diversa, come un `HashSet` per migliorare le prestazioni.

   Ora lo sviluppatore sarà costretto a implementare un adattatore che converte i dati dal database (o dal `HashSet`) in una lista, solo per rispettare l'interfaccia pubblica esistente.

2. **Implicazioni sulle prestazioni**

   Restituire una lista potrebbe diventare un'operazione costosa. Ad esempio:

   - Se i dati sono memorizzati in un database, sarà necessario eseguire una query per recuperare tutti gli elementi.
   - Se i dati sono in una struttura non sequenziale (come un grafo), sarà necessario fare una conversione in una lista.

3. **Rigidità dell'API**

   Il metodo `getElements` comunica un'idea precisa: il cliente dell'API si aspetta sempre una lista. Cambiare questa aspettativa (ad esempio, restituendo un

oggetto diverso) diventa molto difficile senza rompere il codice esistente che usa l'API.

Come evitare il problema?

Per evitare di vincolare l'implementazione futura, è meglio progettare API che espongano solo ciò che è **necessario** per rispettare il contratto della classe, lasciando liberi i dettagli dell'implementazione.

Strategie pratiche:

1. **Esporre un'interfaccia più astratta**

   Invece di restituire una lista concreta, si può restituire un'astrazione più generica, come un `Stream` o un `Iterable`:

   ```
   public Iterable<Integer> getElements() {
       return elements;  // Restituisce un'iterabile
   }
   ```

   Con questa scelta:

   - L'implementazione interna rimane flessibile: si può passare da una `List` a un `Set`, a un database o qualsiasi altra cosa che supporti l'iterazione.
   - Non si fanno assunzioni su come il cliente userà i dati (esempio: il cliente può convertirli in una lista solo se ne ha bisogno).

2. **Fornire metodi specifici per operazioni comuni**

   Invece di esporre direttamente tutti i dati, è possibile progettare metodi che soddisfano le esigenze specifiche degli utenti. Ad esempio:

   ```
   public boolean contains(int element) {
       return elements.contains(element);  // Verifica se l'elemento è presente
   }

   public int size() {
   ```

```
    return elements.size();  // Restituisce il numero di elementi
}
```

In questo modo, si riduce la necessità di esporre l'intera rappresentazione.

3. **Documentare il comportamento, non l'implementazione**

   La documentazione del metodo dovrebbe descrivere **cosa fa** il metodo (es. "restituisce tutti gli elementi dell'insieme") e non come è implementato (es. "restituisce una lista"). Questo comunica agli utenti che l'implementazione interna potrebbe cambiare.

Astrazione dell'Iterazione (Chapter 6)

Data una collezione di elementi, una delle operazioni più comuni da fare è "scorrere" tra essi, cioè una cosa del tipo

```
for all elements of the set
    do action
```

Prendiamo per esempio la classe

```
public class IntSet {
    // overview: IntSets are mutable, unbounded sets of integers.
    // A typical IntSet is {x1, ... , xn}.
    // constructors
    public IntSet ( )
    // effects: Initializes this to be empty.
    // methods
    public void insert (int x)
    // modifies: this
    // effects: Adds x to the elements of this, i.e., this_post = this + { x }.
    public void remove (int x)
    // modifies: this
    // effects: Removes x from this, i.e., this_post = this-{x}
    public boolean isIn (int x)
```

```
    // effects: If x is in this returns true else returns false.
    public int size ( )
    // effects: Returns the cardinality of this.
    public int choose ( ) throws EmptyException
    // effects: If this is empty, throws EmptyException else
    // returns an arbitrary element of this.
}
```

Al di fuori della classe, non ho un modo “comodo” per scorrere l’intset.

Se volessi per esempio, calcolare la somma degli elementi nell’intSet? Ho a disposizione solo insert, remove, isIn, size, choose (che restituisce un elemento “random” dell’intSet)

Potrei implementare il metodo “somma” all’interno della classe, ma se poi invece volessi scorrere l’IntSet per fare una ricerca sequenziale? Implemento search(), poi se volessi scorrere per calcolare la media?

Non possiamo immaginare di implementare ogni metodo possibile all’interno della classe.

Dobbiamo implementare questi metodi, al di fuori della classe.

Non posso esporre l’array che implementa IntSet, altrimenti l’utente potrebbe modificarlo a piacimento, invalidando la Rep. Invariant (inserendo per esempio, valori duplicati, senza passare per il metodo “insert”, che controlla la presenza di duplicati).

Iteratore

Per supportare l’iterazione, devo far si che si possa “scorrere” la collezione di elementi, senza però “distruggerla”.

Implemento un **iteratore**, un metodo della classe che restituisce un oggetto **generatore**, con due metodi:

- boolean hasNext()
- T next() throws NoSuchElementException

Il generatore, implementa il contratto (interfaccia, serie di metodi) Iterator<>

> L’iteratore, è un metodo della classe, che restituisce un oggetto generatore, che implementa il contratto Iterator<>, e i suoi metodi, in particolare i metodi hasNext() e next()

```
package it.unimi.di.prog2.h10;

import it.unimi.di.prog2.h08.impl.EmptyException;
import java.util.ArrayList;
import java.util.List;

/**
 * {@code IntSet}s are mutable, unbounded sets of integers.
 *
 * <p>A typical IntSet is \( S = \{x_1, \ldots, x_n \} \).
 */
public class IntSet implements Iterable<Integer> {

  // Fields

  /** The {@link List} containing this set elements. */
  private final List<Integer> els;

  // Constructors

  /**
   * Initializes this set to be empty.
   *
   * <p>Builds the set \( S = \varnothing \).
   */
  public IntSet() {
```

```
    els = new ArrayList<>();
  }

  /**
   * A *copy constructor*.
   *
   * @param other the {@code IntSet} to copy from.
   */
  public IntSet(IntSet other) {
    els = new ArrayList<>(other.els);
  }

  // Methods

  /**
   * Looks for a given element in this set.
   *
   * @param x the element to look for.
   * @return the index where {@code x} appears in {@code els} if the element
belongs to this set, or
   *     -1
   */
  private int getIndex(int x) {
    return els.indexOf(x);
  }

  /**
   * Adds the given element to this set.
   *
   * <p>This method modifies the object, that is: \( S' = S \cup \{ x \} \).
   *
   * @param x the element to be added.
   */
  public void insert(int x) {
    if (getIndex(x) < 0) els.add(x);
  }
```

```
/**
 * Removes the given element from this set.
 *
 * <p>This method modifies the object, that is: \( S' = S \setminus \{ x \} \).
 *
 * @param x the element to be removed.
 */
public void remove(int x) {
  int i = getIndex(x);
  if (i < 0) return;
  int last = els.size() - 1;
  els.set(i, els.get(last));
  els.remove(last);
}

/**
 * Tells if the given element is in this set.
 *
 * <p>Answers the question \( x\in S \).
 *
 * @param x the element to look for.
 * @return whether the given element belongs to this set, or not.
 */
public boolean isIn(int x) {
  return getIndex(x) != -1;
}

/**
 * Returns the cardinality of this set.
 *
 * <p>Responds with \( |S| \).
 *
 * @return the size of this set.
 */
public int size() {
```

```
    return els.size();
  }

  /**
   * Returns an element from this set.
   *
   * @return an arbitrary element from this set.
   * @throws EmptyException if this set is empty.
   */
  public int choose() throws EmptyException {
    if (els.isEmpty()) throw new EmptyException("Can't choose from an empty
set");
    return els.get(els.size() - 1);
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof IntSet other)) return false;
    if (els.size() != other.els.size()) return false;
    for (int e : els) if (!other.isIn(e)) return false;
    return true;
  }

  @Override
  public int hashCode() {
    int result = 0;
    for (int e : els) result += e; // This is a very bad hash function!
    return result;
  }

  @Override
  public String toString() {
    String lst = els.toString();
    return "IntSet: {" + lst.substring(1, lst.length() - 1) + "}";
  }
```

```
    /**
  * Returns a generator for this class
  */

  public Iterator<Integer> iterator(){
      return new IntGenerator(els)
  }
}
```

Andiamo ora a implementare la classe IntGenerator, di cui ne restituiremo un oggetto (l'iteratore).

```
/**
* Generatore che restituisce tutti gli elementi di un insieme,
*implementa interfaccia "Iterator"
*/
public class IntGenerator implements Iterator<Integer>{

    /** Lista degli elementi dell'IntSet*/
    private final List<Integer> els;
    /**Indice a cui sono arrivato nello scorrimento*/
    private int index;


    /*-
    *AF: l'iteratore restituirà els.get(index) a meno che index>=els.size()
    *RI: els non è null e non contiene null, index è compreso tra 0 e els.size()

    */

    public IntGenerator(List<Integer> els) throws IllegalArgumentException{
        if (els==null){
            throw new IllegalArgumentException("List can't be null");
        for (Integer element i: els){
            if (e==null){
```

```
            throw new IllegalArgumentException("List can't contain null element
s");
        }
      }
      this.els=els;
   }

   @Override
   public boolean hasNext(){
      return index < els.size();
   }

   @Override
   public Integer next(){
      if(hasNext()){
         int next = els.get(index);
         index++;
         return next
      }else{
      throw new NoSuchElementException("No more elements in IntSet")
   }

}
```

Ora potrò utilizzare l'iteratore, in questo modo:

```
public class IntSetUser{
   IntSet set = new IntSet();
   set.insert(2);
   set.insert(3);
   set.insert(4);

   Iterator<Integer> iteratoreSet = set.iterator();

   while(iteratoreSet.hasNext()){
      System.println(iteratoreSet.next());
```

```
    }

    //OPPURE, siccome IntSet è iterable:

    for(int element : set){
        System.out.println(element);
    }
}
```

Classi innestate

Java permette di definire classi dentro altre classi:

```
class OuterClass {
    ...
    class InnerClass { //non-static nested class
        ...
    }
    static class StaticNestedClass { //static nested class
        ...
    }
}
```

Classi annidate non statiche, hanno accesso a tutti gli attributi della classe esterna (anche se privati)

Classi annidate statiche, non accedono agli attributi della classe esterna.

Ha senso annidare classi, se la classe interna è utile solo alla classe esterna, per scrivere codice più pulito.

Le classi interne, esistono solo all'interno di istanze della classe esterna, pertanto per istanziare la classe interna, bisogna prima istanziare quella esterna:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Le classi interne statiche invece, possono essere istanziate direttamente (se siamo all'interno della stessa outer class):

```
public class OuterClass {

    String outerField = "Outer field";
    static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
            // Compiler error: Cannot make a static reference to the non-static
            //     field outerField
            // System.out.println(outerField);
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }

    public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

```
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("--------------------");
        StaticNestedClass staticNestedObject = new StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("--------------------");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);
    }
}
```

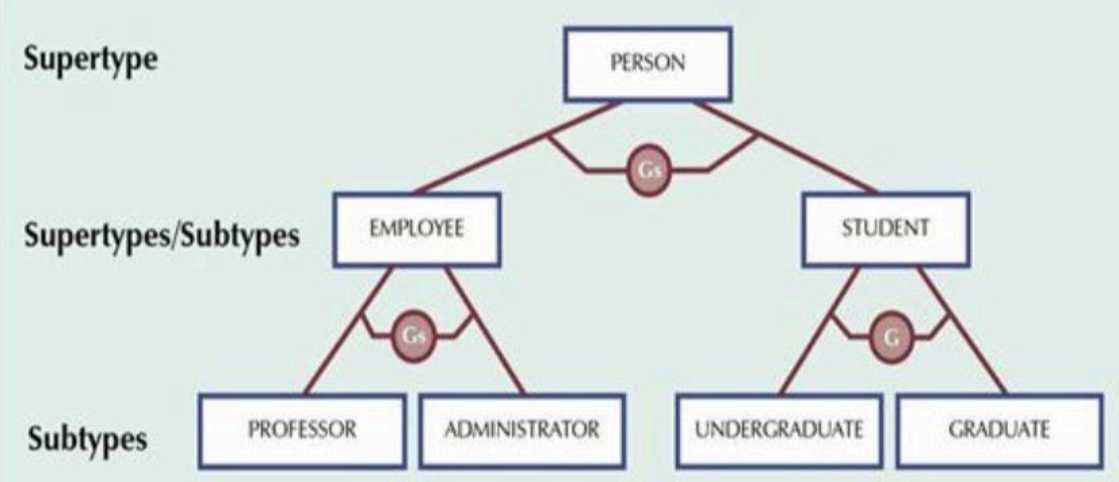
Le classi esterne, devono necessariamente essere pubbliche.

Gerarchia dei tipi [Chapter 7]

Dal concetto di Supertipo e Sottotipo, abbiamo capito che tutti i sottotipi ereditano i metodi del supertipo, e le chiamate a questi metodi si comportano in modo simile.

I sottotipi poi, possono estendere il comportamento dei metodi generici del supertipo (overriding) o implementarne di nuovi.

Una gerarchia di tipi rappresenta una famiglia di tipi, alla cui cima vi è un supertipo la cui specifica è comune a tutti i membri (sottotipi) della famiglia:



Secondo il principio di sostituzione (LSP, Liskov Substitution Principle), il comportamento del supertipo deve essere supportato dal sottotipo: il codice scritto per un supertipo deve funzionare anche se eseguito su un sottotipo

Assegnamento

Se S è un sottotipo di T, posso assegnare oggetti di s a variabili di tipo T.

> For example, suppose that DensePoly and SparsePoly are subtypes of
> Poly. (The idea is that DensePoly provides a good implementation of Polys
> that have relatively few zero coefficients below the degree term, and SparsePoly is good for the Polys that don’t match this criterion.)
>
> Then the following code is permitted:
> `Poly p1 = new DensePoly( ); // the zero Poly`
> `Poly p2 = new SparsePoly(3, 20); // the Poly 3×20.`

Capiamo quindi, che sono “legali” assegnamenti in cui l’oggetto assegnato è di tipo diverso dal tipo della variabile.

In questi casi, distinguiamo due tipi:

- Tipo apparente della variabile (apparent type): è ciò che il compilatore vede al momento della dichiarazione della variabile (Poly)
- Tipo concreto della variabile (actual type): ciò che l’oggetto assegnato è davvero (DensePoly o SparsePoly), ricavato a runTime.

L’actual type, sarà sempre un sottotipo dell’apparent type.

Il compilatore, esegue il type checking basandosi solo sull’apparent type.

In particolare, capisce quali chiamate a metodi sono legali e quali no, solo in base al tipo apparente

```
int d = p1.degree( );
```

Il compilatore sa che è legale, perchè il tipo `Poly` ha un metodo `degree()`

Ma supponiamo che Poly non abbia un metodo degree, e che degree sia implementato solo nel sottotipo DensePoly, allora il compilatore non mi permetterebbe di richiamare il metodo.

Se scrivo il codice “in termini di supertipo” posso utilizzare solo i metodi del supertipo, se invece dichiaro la mia variabile come `DensePoly p1 = new DensePoly( ); // the zero Poly`, potrò usare anche i metodi extra del sottotipo.

Dispatching

Meccanismo con cui il compilatore determina la segnatura del metodo da invocare, e viene individuata l’implementazione da eseguire (JVM).

Il caso più elementare è quello di una singola classe in cui ci sia un solo metodo con un dato nome:

```
public class Simple {
  public void f() {
  System.out.println("Simple::f");
 }
}
```

In queste circostanze, è evidente che il tipo *apparente* e *concerto* non possono che coincidere e che l'invocazione di `f` su un oggetto di tipo `Simple` non può che produrre l'invocazione dell'unica implementazione esistente.

```
Simple s = new Simple();
s.f();
```

```
Simple::f
```

Overloading

Se vengono definiti più metodi col medesimo nome, ossia c'è un di *overloading*, la selezione della segnatura del metodo da invocare segue la logica del minor numero di conversioni (il compilatore tenta di individuare la segnatura detta *most specific*).

```
public class Overload {
  public void f(int x) {
  System.out.println("Overload::f(int)");
 }
  public void f(double x) {
  System.out.println("Overload::f(double)");
 }
}
```

In alcuni casi il numero di conversioni è zero

```
Overload o = new Overload();
o.f(1);
o.f(1.0); // 1.0 è un double, nessuna conversione
```

```
Overload::f(int)
```

```
Overload::f(double)
```

mentre in altri è sufficiente effettuare una conversione (da `float` a `double`) e non ci sono altre possibilità

```
o.f(1.0f); // 1.0f è un float, una conversione
```

```
Overload::f(double)
```

Le cose si complicano se, date le segnature dei metodi e i tipi dei parametri concreti nell'invocazione, esiste più di una segnatura che si adatterebbe alla chiamata a parità di numero di conversioni.

```
public class Overload {
  public void f(int x, double y) {
  System.out.println("Overload::f(int, double)");
 }
  public void f(double x, int y) {
  System.out.println("Overload::f(double, int)");
```

```
  }
}
```

Usando due `int` entrambe i metodi sono invocabili con una conversione a `double` (del primo, o del secondo argomento)

```
Overload o = new Overload();
o.f(1, 1);
```

```
| o.f(1, 1);reference to f is ambiguous  both method f(int,double) in Overload and method f(double,int) in Overload match
```

In questo caso non è possibile individuare la segnatura *most specific*, il compialtore non può scegliere quale segnatura selezione!

È utile ribadire che ciò dipende dal tipo di invocazione: è evidente che le invocazioni che non causano conversioni, ad esempio, sono entrambe legittime.

```
o.f(1, 1.0);
o.f(1.0, 0);
```

```
Overload::f(int, double)
```

```
Overload::f(double, int)
```

Ereditarietà

L'introduzione di un sottotipo apre, tra le altre, la possibilità che su un oggetto di un sottotipo venga invocato un metodo definito nel supertipo.

```
public class Above {
  public void f() {
  System.out.println("Above::f");
  }
}

public class Below extends Above {
  public void g() {
  System.out.println("Below::g");
  }
}
```

Per prima cosa, osserviamo che, considerando anche i sottotipi, il tipo apparente e concreto non necessariamente coincidono: si aprono tre possibilità a seconda che il tipo apparente e concreto siano, rispettivamente

- `Above` e `Above`,
- `Above` e `Below`,
- `Below` e `Below`

evidentemente il caso `Below` e `Above` non è possibile in quanto il secondo non è sottotipo del primo.

Nel primo caso è possibile solo l'invocazione di `f`

```
Above aa = new Above();
aa.f();
```

```
Above::f
```

ma non quella di `g`, dato non è definita in tale tipo

```
aa.g();
```

```
| aa.g();cannot find symbol  symbol:   method g()
```

Nel secondo caso, è possibile invocare `f` perché è visibile (a partire dal tipo apparente) e la sua implementazione (nel tipo concreto) viene ereditata dal supertipo

```
Above ab = new Below();
ab.f();
```

```
Above::f
```

Sebbene nel tipo concreto sia definita `g`, il compilatore sceglie il metodo sulla base del tipo apparente, quindi la chiamata di `g` resta impossibile.

```
ab.g();
```

```
| ab.g();cannot find symbol  symbol:   method g()
```

Nel caso del sottotipo, invece, `g` è visibile poiché è definito in tale tipo (e `f` lo è perché ereditata), quindi sono possibili entrambe le invocazioni.

```
Below bb = new Below();
bb.f();
bb.g();
```

```
Above::f
```

```
Below::g
```

Overriding

Un sottotipo può decidere di riscrivere l'implementazione di un metodo ereditato, ossia farne l'*overriding*.

```
public class Above {
   public void f(double x) {
   System.out.println("Above::f(double)");
  }
}

public class Below extends Above {
```

```
  @Override
    public void f(double x) {
   System.out.println("Below::f(double)");
  }
}
```

Affinché ciò avvenga, è però necessario che il metodo riscritto abbia la medesima segnatura di quello nel supertipo.

Usando l'annotazione `@Override`, che serve ad esprimere l'intenzione del programmatore, il compilatore può accorgersi e segnalare, nel caso in cui la segnatura fosse diversa, che la nuova implementazione non è davvero un *override* (ma solo un *overload* come sarà chiarito nella prossima sezione)!

```
public class BelowErr extends Above {
 @Override
    public void f(int x) {
   System.out.println("BelowErr::f(int)");
  }
}
```

```
|  @Overridemethod does not override or implement a method from a supertype
```

Tornando a considerare i tre possibili casi di combinazione tra tipo apparente e concreto

```
Above aa = new Above();
Above ab = new Below();
Below bb = new Below();
```

è ovvio che su `aa` e `bb` l'invocazione corrisponderà alle implementazioni definite nella classe del tipo (apparente e concreto):

```
aa.f(1.0);
bb.f(1.0);
```

```
Above::f(double)
```

```
Below::f(double)
```

Il caso interessante è quello in cui il tipo concreto è il sottotipo; in tal caso, una volta che il compilatore ha determinato che la segnatura da chiamare è `f(double)`, l'invocazione riguarderà però il codice presente nell'implementazione del tipo concreto:

```
ab.f(1.0);
```

```
Below::f(double)
```

Overloading ed ereditarietà

Un caso più complesso (e interessante) è quando l'*overloading* è determinato dall'ereditarietà, ossia quando è un metodo del sottotipo a causare l'overloading di uno che è definito nel supertipo.

```
public class Above {
  public void f(double x) {
    System.out.println("Above::f(double)");
  }
}

public class Below extends Above {
  public void f(int x) {
    System.out.println("Below::f(int)");
  }
}
```

Come nel caso precedente, i casi in cui il tipo apparente coincide con quello concreto e non ci sono conversioni, sono banali:

```
Above aa = new Above();
Below bb = new Below();

aa.f(1.0);
bb.f(1);
```

```
Above::f(double)
```

```
Below::f(int)
```

Nel caso del supertipo, la chiamata con argomento `int` seleziona l'unico metodo presente (la cui segnatura è compatibile grazie ad una conversione)

```
aa.f(1);
```

```
Above::f(double)
```

In quello del sottotipo, la chiamata con argomento `double` seleziona il metodo ereditato (che non richiede conversioni)

```
bb.f(1.0);
```

```
Above::f(double)
```

La cosa si fa interessante se il tipo apparente non coincide con quello concreto. In tal caso, non sorprendentemente, la chiamata con argomento `double` seleziona il metodo del tipo apparente con tale segnatura:

```
Above ab = new Below();

ab.f(1.0);
```

```
Above::f(double)
```

Cosa succede però con argomento `int`? Dal momento che il sottotipo ha un metodo che non richiede conversioni, ci si potrebbe attendere che sia esso a venir eseguito.

```
ab.f(1);
```

```
Above::f(double)
```

Questo però non avviene perché il compilatore seleziona la segnatura sulla base del tipo apparente: per `Above` la segnatura selezionata è `f(double)` che è compatibile grazie ad una conversione. Una volta selezionata la segnatura, l'invocazione utilizzerà l'implementazione di un metodo di tale segnatura nel sottotipo; tale metodo non è definito nel sottotipo, ma è ereditato dal supertipo.

Overriding e overloading (e ereditarità)

Avendo analizzato separatamente i vari meccanismi che regolano i casi precedenti, non è difficile comprendere un caso in cui si presentino assieme tutte le possibilità.

```
public class Above {
  public void f(double x) {
  System.out.println("Above::f(double)");
 }
}

public class Below extends Above {
  public void f(int x) {
  System.out.println("Below::f(int)");
 }
 @Override
  public void f(double x) {
  System.out.println("Below::f(double)");
 }
}
```

Restano come sempre i casi banali (tipo apparente coincidente con il concreto, nessuna conversione):

```
Above aa = new Above();
Below bb = new Below();

aa.f(1.0);
bb.f(1);
bb.f(1.0);
```

```
Above::f(double)
```

```
Below::f(int)
```

`Below::f(double)`

Nel caso del supertipo, l'invocazione col tipo `int` sarà soddisfatta tramite una conversione

`aa.f(1);`

`Above::f(double)`

Nel caso in cui il tipo apparente è diverso dal concreto, quale che sia il tipo dell'argomento, verrà selezionata l'unica segnatura possibile dato il tipo apparente che è `f(double)`. Certamente, una volta selezionata la segnatura, l'invocazione si userà però l'implementazione del sottotipo che fa overriding di quella nel supertipo.

```
Above ab = new Below();

ab.f(1);
ab.f(1.0);
```

`Below::f(double)`

`Below::f(double)`

Su `ab` non c'è quindi verso di ottenere l'esecuzione del metodo definito in `Below` con segnatura `f(int)`, o di alcun metodo di nome `f` definito in `Above`!

> Si sceglie sempre tramite il tipo apparente, tranne in caso di overriding, in quel caso, se possibile, si sceglie il metodo del tipo reale (sottotipo).

Definire gerarchie

Le gerarchie di tipo, sono definite in Java mediante il meccanismo dell'ereditarietà.

Ciò permette di creare sottoclassi di altre classi (superclassi).

Ci sono due tipi di classi in Java:

- Classi concrete, forniscono implementazione completa del tipo.
- Classi astratte, forniscono implementazione parziale, non hanno oggetti associati. Potrebbero avere metodi astratti, cioè metodi non implementati nella superclasse, e che andranno implementati nelle sottoclassi.

Una sottoclasse dichiara la propria superclasse utilizzando la keyword extends: ciò permette alla sottoclasse di ereditare tutti i metodi della superclasse.

La sottoclasse, eredita tutti i metodi **final** e **normali**, deve implementare i metodi **astratti** della superclasse, può riscrivere (override) i metodi **non final** della superclasse (i metodi final, non possono essere reimplementati dalle sottoclassi).

La rep di una sottoclasse, è composta dai propri attributi, e quelli della superclasse.

La sottoclasse accederà direttamente agli attributi della superclasse solo se essi sono dichiarati come protected (altrimenti, accederà agli attributi tramite i metodi mutazionali del padre).

E' bene evitare il più possibile che le sottoclassi accedano agli attributi della superclasse direttamente: è meglio farlo mediante i metodi mutazionali, così che in futuro possa cambiare la rappresentazione della superclasse, senza dover cambiare anche quella della sottoclasse.

```
public class MaxIntSet extends IntSet {

  /** The biggest element, if set is not empty */
  private int biggest;

  // RI: size() == 0 or isIn(biggest) and for every x isIn(x) implies biggest >= x.
  // AF: coincides with that of IntSet

  /** Construct an empty {@code MaxIntSet}. */
```

```
  public MaxIntSet() {
    super(); //per costruire un emptyset, semplicemente richiama il costruttore
del padre
  }

  @Override
  public void insert(final int x) {
    if (size() == 0 || x > biggest) biggest = x;
    super.insert(x); //metodo insert del padre
  }

  @Override
  public void remove(final int x) {
    super.remove(x); //remove del padre
    if (size() == 0 || x != biggest)
      return; // observe that if x > biggest it was not actually in this, so we don't
need to
    // update biggest
    biggest = Integer.MIN_VALUE;
    for (int z : this) if (z > biggest) biggest = z;
  }

  /**
   * Returns the maximum value in the set, or raises {@link EmptyException} ot
herwise.
   *
   * @return the maximum value in the set.
   * @throws EmptyException if the set is empty.
   */
  public int max() throws EmptyException {
    if (size() == 0) throw new EmptyException();
    return biggest;
  }

  /**
   * Checks the Representation Invariant.
```

```
   *
   * @return true if and only if the Representation Invariant holds.
   */
  public boolean repOk() {
    // no need to check super.repOk() because there is no state sharing
    if (size() == 0) return true;
    boolean found = false;
    for (int z : this) {
      if (z > biggest) return false;
      if (z == biggest) found = true;
    }
    return found;
  }

  @Override
  public String toString() {
    if (size() == 0) return "Max" + super.toString();
    else return "Max" + super.toString() + (size() > 0 ? ", max = " + biggest : "");
  }
}
```

La keyword super, permette di riferirsi alla classe padre.

La AF del figlio, si scrive riferendosi all'AF del padre. In questo caso:

```
// The abstraction function is:
// AF_MaxIntSet(c) = AF_IntSet(c)
```

Il fatto che abbia aggiunto un attributo "biggest", non cambia ciò che MaxIntSet rappresenta, pertanto la sua AF è identica a quella del padre.

La rep. invariant invece è

```
// the rep invariant is:
// I_MaxIntSet(c) = c.size > 0 ⇒
// (c.biggest in AF_IntSet(c) &&
// for all x in AF_IntSet(c) (x <= c.biggest))
```

Anche qui, mi riferisco al set utilizzando la AF del padre.

Non bisogna riscrivere i contratti di insert e remove poiché questi li ereditano dal supertipo e non cambiano, sì vanno ad aggiornare biggest ma questo non deve interessare all'utente, non tocca il contratto. Va a cambiare solo l'implementazione.

x viene dichiarato final sia nel metodo insert che remove perché non c'è bisogno di modificare il valore che viene passato in input. È utile per evitare errori in cui per esempio, per sbaglio si riassegna il valore di x

Ereditarietà a carattere ontologico

Una delle motivazioni con cui il concetto di *ereditarietà* viene presentato (e spesso l'unico genere di esempio che lo riguarda) ha a che fare con entità che, nella realtà, sono dal punto di vista ontologico in una relazione di *più specifico*/*meno specifico* o di "*è un*" (*is a*), ossia in cui l'ereditarietà è vista dal punto di vista ontologico.

Uno degli esempi più classici è tratto dalla geometria e si basa sul fatto che le varie figure piane (cerchi, rettangoli, quadrati...) sono tutte esempi di un'entità più generica che è, appunto, la figura piana.

In tale contesto si fa spesso l'esempio del **quadrato** come sottotipo di **rettangolo** (in simboli *quadrato* << *rettangolo*): tutti i quadrati, infatti, sono a ben vedere dei rettangoli (che hanno la base uguale all'altezza), ma non è vero il viceversa.

Nell'ambito dei *tipi di dato astratti* però, abbiamo visto che l'*estensione* è di solito sviluppata nella direzione opposta: il rettangolo ha infatti un attributo in più

(l'altezza) del quadrato (che è identificato dalla sola base).

Risulta molto naturale infatti scrivere il tipo `Square` con un solo attributo

```
public class Square {
  private int base;

  public Square(int base) {
    this.base = base;
  }

  public int base() {
    return this.base;
  }

  public void base(int base) {
    this.base = base;
  }

  @Override
  public String toString() {
    return String.format("Square, base = %d", base);
  }
}
```

[sorgente]

e definire per estensione il tipo `Rectangle`

```
public class Rectangle extends Square {
  private int height;

  public Rectangle(int base, int height) {
    super(base);
    this.height = height;
  }

  public int height() {
    return this.height;
  }

  public void height(int height) {
    this.height = height;
  }

  @Override
  public String toString() {
    return String.format("Rectangle: base = %d, height = %d", base(), height);
  }
}
```

[sorgente]

Ma in questo modo la relazione di sottotipo `Rectangle` << `Square` è l'opposto della relazione ontologica per cui *quadrato* << *rettangolo*!

Si potrebbe certamente pensare che questa inversione sia fittizia e sia legata alla scelta implementativa; si potrebbe infatti pensare che è possibile ribaltare la direzione in cui un tipo estende l'altro pur di *sovrascrivere* i metodi mutazionali del rettangolo in modo che, nel caso del quadrato, adeguino la base qualora venga cambiata l'altezza.

Si potrebbe cioè pensare di porre `Rectangle` come supertipo

```
public class Rectangle {
  private int base, height;

  public Rectangle(int base, int height) {
    this.base = base;
    this.height = height;
  }

  public int base() {
    return this.base;
  }

  public void base(int base) {
    this.base = base;
  }

  public int height() {
    return this.height;
  }

  public void height(int height) {
    this.height = height;
  }

  @Override
  public String toString() {
    return String.format("WrongFigures.Rectangle: base = %d, height = %d", base(), height);
  }
}
```

[sorgente]

con due metodi mutazionali per base e altezza, e implementare `Square` per estensione, facendo attenzione a sovrascrivere i metodi mutazionali `base` e `height` in modo che mantengano coerenti base e altezza

```
public class Square extends Rectangle {
```

```
    public Square(int base) {
        super(base, base);
    }

    public void base(int base) {
        super.base(base);
        super.height(base);
    }

    public void height(int height) {
        super.base(height);
        super.height(height);
    }

    @Override
    public String toString() {
        return String.format("Square, base = %d", base());
    }
}
```

[sorgente]

Il principio di sostituzione

Immaginiamo ora di voler costruire un istogramma di rettangoli in cui i rettangoli aggiunti all'istogramma sono mantenuti in ordine crescente d'altezza. Supponiamo inoltre che per ragioni "tipografiche" sia sensato alterare la base dei rettangoli anche una volta che sono parte dell'istogramma (ad esempio, per restringerne l'ampiezza complessiva).

```
public class Histogram {

    List<Rectangle> rectangles = new LinkedList<>();

    public void add(Rectangle r) {
        int i;
        for (i = 0; i < rectangles.size(); i++) if (rectangles.get(i).height() > r.height()) break;
        rectangles.add(i, r);
    }

    public void changeBase(Rectangle o, int base) throws NoSuchElementException {
        final int idx = rectangles.indexOf(o);
        if (idx != -1) rectangles.get(idx).base(base);
        else throw new NoSuchElementException();
    }

    @Override
    public String toString() {
        return rectangles.toString();
```

```
    }
}
```

[sorgente]

Evidentemente, ciò che accade se sostituissimo il sottotipo `Square` al posto di `Rectangle` il metodo `changeBase` avrebbe l'effetto (inatteso) di modificare anche l'altezza, non preservando il contratto di `Histogram`. Si consideri il seguente *client*

```
public class MainSR {

    public static void main(String[] args) {

        Histogram hist1 = new Histogram();
        Rectangle r1 = new Rectangle(4, 4), r2 = new Rectangle(3, 3);
        hist1.add(r1);
        hist1.add(r2);
        System.out.println(hist1);
        hist1.changeBase(r2, 6);
        System.out.println(hist1);

        Histogram hist2 = new Histogram();
        Square s1 = new Square(4), s2 = new Square(3);
        hist2.add(s1);
        hist2.add(s2);
        System.out.println(hist2);
        hist2.changeBase(s2, 6);
        System.out.println(hist2);
    }
}
```

[sorgente]

L'uso in `hist1` mostra il comportamento inteso, ma non è così per quello di `hist2`, come risulta dall'esecuzione del codice, che genera l'output

```
['[WrongFigures.Rectangle: base = 3, height = 3, WrongFigures.Rectangle: base = 4, height = 4]',
'[WrongFigures.Rectangle: base = 6, height = 3, WrongFigures.Rectangle: base = 4, height = 4]',
'[Square, base = 3, Square, base = 4]',
'[Square, base = 6, Square, base = 4]']
```

Come si nota, infatti, i rettangoli restano ordinati per altezza anche dopo il cambiamento della base (prime due righe), ma così non è per i quadrati, che, evidentemente, dopo il cambiamento di base hanno anche le altezze in ordine decrescente (ultime due righe).

Da questo testo emerge un problema fondamentale nella progettazione object-oriented noto come il "Square-Rectangle Problem" o "Circle-Ellipse Problem". Il

dilemma nasce dal conflitto tra la relazione ontologica (matematica) e il principio di sostituzione di Liskov (LSP).

> if f(x) is a property provable about objects x of type T, then f(y) should be true for objects y of type S where S is a subtype of T.

An object can be replaced by a sub-object without breaking the program, as what holds for T-objects holds for S-objects *is what must be understood.*

A square breaks that principle since it doesn't have a different `height` and `width`. So not everything true for `Rectangle` is true for `Square`.

Matematicamente, sappiamo che "ogni quadrato è un rettangolo" (quadrato < rettangolo). Tuttavia, dal punto di vista della programmazione orientata agli oggetti, questa relazione crea problemi perché viola il Principio di Sostituzione di Liskov.

Il testo mostra chiaramente che:

1. Se facciamo `Square extends Rectangle`, violiamo LSP perché il quadrato deve mantenere base = altezza, modificando comportamenti che il client del rettangolo non si aspetta (come visto nell'esempio dell'Histogram)
2. Se facciamo `Rectangle extends Square`, violiamo la relazione ontologica naturale

La soluzione corretta in questo caso è NON utilizzare l'ereditarietà tra quadrato e rettangolo. Invece, dovremmo:

1. Creare un'interfaccia comune (es. `Shape` o `Quadrilateral`)
2. Implementare `Square` e `Rectangle` come classi separate che implementano questa interfaccia
3. Se necessario, utilizzare la composizione invece dell'ereditarietà

Ecco un esempio di implementazione corretta:

```
interface Shape {
    int getArea();
```

```
    int getPerimeter();
    // altri metodi comuni
}

class Rectangle implements Shape {
    private int base;
    private int height;

    public Rectangle(int base, int height) {
        this.base = base;
        this.height = height;
    }

    public void setBase(int base) {
        this.base = base;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    // implementazione metodi Shape
}

class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public void setSide(int side) {
        this.side = side;
    }
```

```
    // implementazione metodi Shape
}
```

Questa soluzione:

1. Rispetta LSP perché non c'è relazione di ereditarietà
2. Mantiene le classi separate con le loro specifiche responsabilità
3. Evita comportamenti inattesi come quelli mostrati nell'esempio dell'Histogram
4. Permette di evolvere le due classi indipendentemente

In conclusione, anche se matematicamente un quadrato è un rettangolo, dal punto di vista del design object-oriented è meglio trattarli come tipi separati che condividono un'interfaccia comune, piuttosto che forzare una relazione di ereditarietà che porterebbe a violazioni del LSP.

Classi astratte

Una classe astratta, propone una sola implementazione parziale di un tipo. Non si possono istanziare oggetti di una classe astratta. Sono una sorta di template da cui derivare ed estendere altre classi

Deve avere uno o più costruttori, può avere attributi, metodi concreti (di cui è fornita un’implementazione), e metodi astratti (di cui è fornita la sola specifica).

L’implementazione dei metodi concreti, permette di scrivere codice comune a tutte le sottoclassi, in modo da rendere il loro codice più snello e semplice da leggere.

Quando ereditiamo i metodi da una classe astratta, non ci interessa definire una relazione di “più/meno specifico” oppure di tipo “is a”: ci interessa il solo riutilizzo del codice.

Utili quando ci interessa avere uno stato condiviso (variabili protected) tra le classi che estendono la classe astratta.

> Preferire sempre variabili private e l'uso di getter e setter, piuttosto che variabili protected, per proteggere lo stato.

Classe astratta:

```
/**
 * An {@code AbstractIntSet} is a mutable, unbounded set of integers.
 *
 * <p>A typical {@code AbstractIntSet} is \( S = \{x_1, \ldots, x_n \} \).
 */
public abstract class AbstractIntSet implements Iterable<Integer> {

  /** The elements of the set. */
  protected int size; //sarà visibile dalle sottoclassi.

  /*-
   * AF(size) → a set with size elements.
   * RI: size >= 0
   */

  /** Creates an empty set. */
  protected AbstractIntSet() { //sarà utilizzabile dalle sottoclassi.
    size = 0;
  }

  /**
   * Adds the given element to this set.
   *
```

```
 * <p>This method modifies the object, that is: \( S' = S \cup \{ x \} \).
 *
 * @param x the element to be added.
 */
public abstract void insert(int x);

/**
 * Removes the given element from this set.
 *
 * <p>This method modifies the object, that is: \( S' = S \setminus \{ x \} \).
 *
 * @param x the element to be removed.
 */
public abstract void remove(int x);

/**
 * Tells if the given element is in this set.
 *
 * <p>Answers the question \( x\in S \).
 *
 * @param x the element to look for.
 * @return whether the given element belongs to this set, or not.
 */
public boolean isIn(int x) {
//l'utilizzo della keyword this in un foreach, richiama l'iteratore dell'oggetto.
  for (int e : this) if (e == x) return true;
  return false;
}

/**
 * Returns the cardinality of this set.
 *
 * <p>Responds with \( |S| \).
 *
 * @return the size of this set.
 */
```

```
  public int size() {
    return size;
  }

  /**
   * Returns an element from this set.
   *
   * @return an arbitrary element from this set.
   * @throws NoSuchElementException if this set is empty.
   */
  public int choose() throws NoSuchElementException {
    if (size == 0) throw new NoSuchElementException("Can't choose from an e
mpty set");
    return iterator().next();
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof AbstractIntSet other)) return false;
    if (size != other.size) return false;
    for (int e : this) if (!other.isIn(e)) return false;
    return true;
  }

  @Override
  public int hashCode() {
    int result = 0;
    for (int e : this) result += e; // This is a very bad hash function!
    return result;
  }

  @Override
  public String toString() {
    StringJoiner sj = new StringJoiner(", ", "{", "}");
    for (Integer e : this) sj.add(e.toString());
```

```
    return sj.toString();
  }
}
```

Nella classe astratta, potrei dover manipolare la rappresentazione (aggiungere, togliere elementi), senza però sapere che struttura dati andrò ad utilizzare nelle sottoclassi: utilizzo allora i metodi astratti della sottoclasse (insert, remove).

IntSet:

```
public class IntSet extends AbstractIntSet {

  /** The set elements. */
  private final List<Integer> elements;

  /*-
   * AF(elements, size) = { elements.get(0), elements.get(1), ..., elements.get(si
ze - 1) }
   * RI:
   *   - super.RI
   *   - elements != null and does not contain nulls.
   *   - elements doesn't contain duplicates
   *   - size == elements.size
   */

  /** Creates an empty set. */
  public IntSet() {
  //il super() non è necessario in quanto fatto da lui implicitamente.
    this.elements = new ArrayList<>();
  }

  @Override
  public Iterator<Integer> iterator() {
    return Collections.unmodifiableCollection(elements).iterator();
  }
```

```
  @Override
  public void insert(int x) {
    // the use of isIn(x) instead of !elements.contains(x)
    // can take advantage of improved implementations in subclasses
    if (!isIn(x)) {
      elements.add(x);
      size++;
    }
  }

  @Override
  public void remove(int x) {
    if (elements.remove(Integer.valueOf(x))) size--;
  }
}
```

Bisogna fare l'override di tutti i metodi astratti del supertipo.

Nella RI, mi riferisco a quella del padre, e inoltre aggiungo riferimenti specifici alla struttura dati scelta.

OrderedIntSet:

```
/**
 * A concrete sorted set of integers.
 *
 * <p>The iterator of this set returns the elements in ascending order.
 */
public class OrderedIntSet extends AbstractIntSet {

  /** The set elements. */
  private final List<Integer> elements;

  /*-
   * AF(elements, size) = { elements.get(0), elements.get(1), ..., elements.get(si
```

```
ze - 1) }
   * RI:
   *   - super.RI
   *   - elements != null and does not contain nulls,
   *   - elements is sorted in ascending order.
   */

  /** Creates an empty set. */
  public OrderedIntSet() {
    this.elements = new ArrayList<>();
  }

  /**
   * Returns the maximum element of this set.
   *
   * @return the maximum element of this set.
   * @throws NoSuchElementException if this set is empty.
   */
  public int max() throws NoSuchElementException {
    if (size == 0) throw new NoSuchElementException("An empty set has no ma
ximum element.");
    return elements.getLast();
  }

  /**
   * Returns the minimum element of this set.
   *
   * @return the maximum element of this set.
   * @throws NoSuchElementException if this set is empty.
   */
  public int min() throws NoSuchElementException {
    if (size == 0) throw new NoSuchElementException("An empty set has no mi
nimum element.");
    return elements.get(0);
  }
```

```
  @Override
  public Iterator<Integer> iterator() {
    return Collections.unmodifiableList(elements).iterator();
  }

  @Override
  public void insert(int x) {
    final int index = Collections.binarySearch(elements, x);
    if (index < 0) {
      elements.add(-index - 1, x);
      size++;
    }
  }

  @Override
  public void remove(int x) {
    if (elements.remove(Integer.valueOf(x))) size--;
  }
}
```

Interfacce

Quando utilizziamo l'astrazione non per definire rapporti ontologici, ma per fornire diverse implementazioni di un supertipo, allora il supertipo sarà una classe astratta o un'interfaccia.

Un'interfaccia, è un insieme di metodi (e costanti), che ogni sottotipo deve implementare.

Tutti i suoi metodi non forniscono implementazione: sono pubblici e astratti.

Quando una classe astratta ha solo metodi astratti, conviene usare un'interfaccia.

Tramite le interfacce, posso definire gerarchie in cui un sottotipo ha più supertipi: una classe può estendere una sola classe, ma può implementare più interfacce.

Le classi implementano interfacce, utilizzando la keyword `extends` nel loro header.

Evoluzioni delle interfacce:

Consideriamo:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

In futuro, vogliamo aggiungere un nuovo metodo all'interfaccia

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

Tutte le classi che implementano l'interfaccia, non funzioneranno più! Dovranno infatti necessariamente implementare `didItWork`.

Inaccettabile.

Posso definire il mio nuovo metodo come default, cioè fornire anche un implementazione.

Tutte le classi che usano l'interfaccia, saranno allora dotate di questa nuova funzionalità, senza modificare il loro codice

```
public interface DoIt {

  void doSomething(int i, double x);
  int doSomethingElse(String s);
  default boolean didItWork(int i, double x, String s) {
     // Method body
  }
}
```

Esempio:

```
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

Vengono aggiunti:

```
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
```

```
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Quando utilizzare interfacce, quando classi astratte?

Caratteristica	Classe astratta	Interfaccia
Stato (variabili di istanza)	Può contenere stato (variabili private o protette).	Non può contenere stato, tranne costanti (`static final`).
Ereditarietà	Può essere estesa da una sola classe concreta (ereditarietà singola).	Una classe può implementare più interfacce (ereditarietà multipla).
Metodi	Può avere sia metodi concreti che astratti.	Può avere metodi astratti e di default (con corpo).
Costruttori	Può avere costruttori per inizializzare lo stato.	Non può avere costruttori.
Accesso	Metodi concreti e astratti possono essere `public` , `protected` , o `package-private` .	Metodi astratti e di default sono implicitamente `public` .
Uso principale	Creare una base comune per classi con comportamento condiviso e stato comune.	Specificare un contratto che definisce comportamenti obbligatori.

Quando scegliere una classe astratta

Usa una classe astratta quando:

1. **C'è stato da condividere**:
    - Se più classi derivate condividono variabili comuni (ad esempio, `name` , `id` , etc.), ha senso utilizzare una classe astratta per gestirle.

- Es: `Animal` che ha una variabile `name` e un metodo concreto `eat()`.

2. **Vuoi fornire una base parzialmente implementata**:
   - Se hai metodi che devono essere definiti dalle sottoclassi (metodi astratti) ma vuoi fornire implementazioni predefinite per altri metodi.
3. **Protezione dello stato**:
   - Le variabili di istanza possono essere dichiarate `protected` per consentire l'accesso alle sottoclassi. Tuttavia, attenzione ai rischi di modifiche indesiderate.

Quando scegliere un'interfaccia

Usa un'interfaccia quando:

1. **Non hai bisogno di stato**:
   - Se il comportamento è definito esclusivamente attraverso metodi e non serve mantenere variabili condivise tra le implementazioni.
2. **Hai bisogno di ereditarietà multipla**:
   - Una classe può implementare più interfacce, consentendo di combinare comportamenti da più fonti.
3. **Flessibilità e compatibilità**:
   - Con i metodi di default, puoi aggiungere nuove funzionalità a un'interfaccia senza rompere le classi esistenti che già la implementano.
4. **Contratto puro**:
   - Quando vuoi definire solo ciò che una classe deve fare (non come farlo). Ad esempio, un'interfaccia `Flyable` che dichiara un metodo `fly()`.

Esempio pratico completo

Interfaccia:

```
java
Copia codice
interface Vehicle {
    void start();
    void stop();
    default void honk() {
        System.out.println("Honking!");
    }
}
```

Classe astratta:

```
java
Copia codice
abstract class PoweredVehicle implements Vehicle {
    protected int fuelLevel;

    PoweredVehicle(int fuelLevel) {
        this.fuelLevel = fuelLevel;
    }

    void refuel(int amount) {
        fuelLevel += amount;
        System.out.println("Refueled: " + amount);
    }
}
```

Classi concrete:

```
java
Copia codice
class Car extends PoweredVehicle {
```

```
    Car(int fuelLevel) {
        super(fuelLevel);
    }

    @Override
    public void start() {
        System.out.println("Car starting with fuel level: " + fuelLevel);
    }

    @Override
    public void stop() {
        System.out.println("Car stopped.");
    }
}

class Bicycle implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bicycle starting.");
    }

    @Override
    public void stop() {
        System.out.println("Bicycle stopped.");
    }
}
```

Conclusione

- Usa **interfacce** per definire contratti e per favorire la flessibilità grazie alla possibilità di ereditarietà multipla.
- Usa **classi astratte** solo quando hai **stato comune o comportamento condiviso** che deve essere riutilizzato.

- La combinazione di entrambe è spesso la scelta più robusta e manutenibile.

Approfondimento su LSP

Il principio di sostituzione prevede che si possa ragionare sul codice solo in base alle specifiche del supertipo: durante l'esecuzione, potrebbero essere utilizzati sottotipi di tale oggetto, il codice deve rimanere valido.

Per garantire ciò, ci sono tre regole:

Signature rule

Un metodo in un sottotipo deve avere la stessa **firma** del metodo nel supertipo affinché sia considerato un **overriding** valido. La firma comprende:

- Nome del metodo.
- Tipo e ordine dei parametri.
- Tipo di ritorno (deve essere lo stesso o un sottotipo del tipo di ritorno dichiarato nel supertipo, concetto noto come **covariant return type**).

Se un metodo nel sottotipo non rispetta questa regola, non sarà considerato un override valido, e il compilatore genererà un errore.

```
class Supertype {
    void insert(Supertype obj) { }
}

class Subtype extends Supertype {
    @Override
    void insert(Supertype obj) { } // OK: firma identica
}
```

```
class Subtype extends Supertype {
   void insert(Subtype obj) { } // Non è un override!
}
```

Questa non è una sovrascrittura valida perché la firma non coincide. Il metodo nel sottotipo accetta solo `Subtype` come parametro, mentre il metodo del supertipo accetta un generico `Supertype`. Di conseguenza:

- Se chiami `insert` su un oggetto di tipo `Subtype`, il metodo chiamato dipenderà dal tipo di riferimento:
  - **Se il riferimento è del tipo `Supertype`**, il compilatore userà `insert(Supertype obj)`, perché è definito nella gerarchia del supertipo.
  - **Se il riferimento è del tipo `Subtype`**, il metodo `insert(Subtype obj)` sarà disponibile.

Methods rule

I metodi di un sottotipo, devono rispettare le specifiche del supertipo.

Questo deve garantire che il codice scritto per il supertipo, continui a funzionare, anche se i metodi chiamati sono quelli del sottotipo.

Le precondizioni di un metodo in un sottotipo, possono essere uguali o più lasche rispetto a quelle dello stesso metodo nel supertipo.

Esempio:

Supertipo:

```
public void addZero()
// requires: this is not empty
// effects: Adds 0 to this
```

Sottotipo:

```
@Override
public void addZero()
// effects: Adds 0 to this
```

La precondizione nel sottotipo, è più debole: ciò va bene, garantisce che il codice scritto per il supertipo, sia compatibile anche per l'uso del sottotipo.

Un sottotipo non può introdurre una precondizione più restrittiva, perché questo potrebbe rendere non valide chiamate al metodo fatte in base alla specifica del supertipo.

Esempio errato:
Supertipo:

```
public void addZero()
// effects: Adds 0 to this
```

Sottotipo:

```
@Override
public void addZero()
// requires: this.size() > 5
// effects: Adds 0 to this
```

In questo caso, il sottotipo introduce una restrizione che il supertipo non aveva, violando la regola della precondizione.

La postcondizione di un metodo nel sottotipo deve essere **più forte** (ovvero garantire almeno ciò che garantisce il supertipo, e possibilmente di più). Questo assicura che il comportamento del sottotipo sia conforme alle aspettative del codice che si basa sul supertipo.

Esempio:

Supertipo:

```
public void insert(int x)
// effects: Adds x to the set
```

Sottotipo:

```
@Override
public void insert(int x)
// effects: Adds x to the set and also logs the addition
```

La postcondizione del sottotipo è più forte, perché include l'aggiunta di `x` al set (come richiesto dal supertipo) e aggiunge ulteriori effetti (registrazione nel log).

Un sottotipo non può avere una postcondizione più debole, perché il comportamento del metodo potrebbe non rispettare ciò che il codice del supertipo si aspetta.

Esempio errato:
Supertipo:

```
public void insert(int x)
// effects: Adds x to the set
```

Sottotipo:

```
@Override
public void insert(int x)
// effects: If x is odd, adds it to the set; else does nothing
```

Il sottotipo non garantisce più che `x` venga aggiunto al set, violando la postcondizione del supertipo.

Properties rule

Il sottotipo, deve mantenere valide tutte le proprietà del supertipo, siano esse:

- Proprietà invarianti: cioè le condizioni della rep invariant, la rep invariant deve essere sempre vera anche nel sottotipo
- Proprietà evoluzionali: descrivono come gli oggetti possono cambiare nel tempo.

Verifica delle invarianti

Per dimostrare che un sottotipo preserva le invarianti del supertipo, bisogna verificare che:

1. **I costruttori del sottotipo** inizializzino gli oggetti rispettando le invarianti.
2. **Tutti i metodi del sottotipo** (sia quelli ereditati che quelli nuovi) preservino le invarianti.

Esempio:
Consideriamo il tipo `FatSet`, che ha l'invariante: **"Un `FatSet` non è mai vuoto"**.
Questo è specificato nell'overview:

```
java
Copia codice
// overview: A FatSet is a mutable set of integers whose size
// is always at least 1.
```

`FatSet` non ha un metodo `remove`, ma ha un metodo `removeNonEmpty`:

```
java
Copia codice
public void removeNonEmpty(int x)
// modifies: this
// effects: If x is in this and this contains other elements,
// removes x from this.
```

Poiché i costruttori di `FatSet` creano sempre un insieme con almeno un elemento e il metodo `removeNonEmpty` non può svuotare l'insieme, l'invariante è rispettata.

Se però introduciamo un sottotipo `ThinSet` con un metodo aggiuntivo `remove`:

```
java
Copia codice
public void remove(int x)
// modifies: this
// effects: Removes x from this.
```

`ThinSet` **non è un sottotipo valido di** `FatSet`, perché il metodo `remove` può svuotare l'insieme, violando l'invariante.

2. Proprietà di evoluzione

Le proprietà di evoluzione descrivono come gli oggetti possono cambiare nel tempo. Ad esempio:

- Un `SimpleSet` può solo **crescere**, ma non può **ridursi**:

```
java
Copia codice
// overview: A SimpleSet is a mutable set of integers.
// SimpleSet objects can grow over time but not shrink.
```

Un sottotipo di `SimpleSet` non può introdurre un metodo come `remove`, perché violerebbe la proprietà di crescita.

Immutabilità come proprietà di evoluzione

Un caso comune di proprietà di evoluzione è l'**immutabilità**. Per un tipo immutabile, gli oggetti non possono cambiare stato nel tempo. Ad esempio, un polinomio `Poly` con grado 6 manterrà sempre lo stesso grado.

Se un tipo è immutabile, i suoi sottotipi generalmente saranno anch'essi immutabili.

Uguaglianza ed ereditarietà

Questa breve nota discute i limiti imposti dall'ereditarietà al soddisfacimento del contratto imposto dal metodo `equals`) della classe `Object` .

La nota segue l'esposizione della *Sezione 7.9.3* del testo "Program Development in Java" (di Barbara H. Liskov *et al.*) e dell'*Item 10* del testo "Effective Java" (di Joshua Bloch).

Simmetria

L'implementazione *ovvia* non funziona: se un sottotipo aggiunge un "valore", non basta controllare con `instanceof` e verificare che siano uguali i valori in gioco...

Consideriamo il seguente tipo `T`

```
public class T {
  private final int a;

  public T(int a) {
    this.a = a;
  }

  public boolean equals(Object obj) {
    if (obj instanceof T other) return a == other.a;
    return false;
  }
}
```

[sorgente]

e il suo sottotipo `S`

```
public class S extends T {
  private final int b;

  public S(int a, int b) {
    super(a);
    this.b = b;
  }

  public boolean equals(Object obj) {
    if (obj instanceof S other) return super.equals(obj) && b == other.b;
    return false;
  }
}
```

[sorgente]

Entrambi i tipi sono dotati di una implementazione di `equals` (qui evidenziata in giallo) che, dopo aver verificato il tipo dell'argomento `obj` procedono all'ovvio controllo dell'identità dei valori del/i campo/i corrispondente/i.

Tale implementazione viola la proprietà di **simmetria**, ossia t~st~s non implica s~ts~t, come dimostrato dal seguente metodo `main`

```
public static void main(String[] args) {
 S s = new S(1, 2);
 T t = new T(1);
 System.out.println(t.equals(s));
 System.out.println(s.equals(t));
}
```

[sorgente]

che produce l'output

```
true
false
```

Quel che accade è che `t instanceof S` è *falso*, ma `s instanceof T` è *vero*, per cui l'implementazione di `equals` di `S` non sa cosa fare con `t`, mentre quella di `T` funziona con `s` (ovviamente trascurando `b`, di cui ignora l'esistenza).

Transitività

Non è facile aggiustare l'implementazione di `equals` di `S` gestendo accortamente il caso in cui `obj` ha il tipo effettivo di `T` imitando il suo comportamento, ossia "scordandosi" di `b`, come (nella parte evidenziata) nel seguente codice

```
public boolean equals(Object obj) {
  if (obj instanceof S other) return super.equals(obj) && b == other.b;
  if (obj instanceof T) return super.equals(obj);
  return false;
}
```

[sorgente]

Così facendo infatti, la simmetria è rispettata, ma risulterà violata la proprietà **transitiva**, ovvero anche avendo s~ts~t e t~ut~u non è detto che s~us~u. Questo è dimostrato dalla seguente porzione di codice

```
public static void main(String[] args) {
 S s = new S(1, 2);
 T t = new T(1);
 S u = new S(1, 3);
```

```
 System.out.println(s.equals(t));
 System.out.println(t.equals(u));
 System.out.println(s.equals(u));
}
```

[sorgente]

che produce l'output

```
 true
true
false
```

Il principio di sostituzione

Si potrebbe pensare che il problema sia `instanceof` che è antisimmetrico, per cui una soluzione che viene talvolta suggerita è di sostituire nell'implementazione di `equals` di `T` la condizione `obj instanceof T` con la condizione `getClass() == obj.getClass()` che è simmetrica (e risulta vera se e solo se gli oggetti hanno identico tipo concreto — attenzione, però, occorre occpuarsi esplicitamente del caso `obj == null`).

```
public boolean equals(Object obj) {
  if (obj == null ) return false ;
  if (getClass() == obj.getClass()) {
   final T t = (T) obj;
   return a == t.a;
 }
  return false ;
}
```

[sorgente]

Si può applicare la stessa scelta anche a `S`, sebbene questo sia utile solo qualora tale classe venisse estesa.

```
public boolean equals(Object obj) {
  if (obj == null ) return false ;
  if (getClass() == obj.getClass()) {
   final S s = (S) obj;
   return super .equals(obj) && b == s.b;
 }
  return false ;
}
```

[sorgente]

In questo modo la simmetria e la transitività sono rispettate perché `equals` funziona solo per oggetti del medesimo tipo, avendo valore `false` per oggetti di tipo diverso

(anche se sottotipi l'uno dell'altro).

Questo però viola il **principio di sostituzione**!

Per cogliere meglio questo aspetto consideriamo `R`, una ulteriore estensione di `T` che non aggiunga alcun "valore", ma solo un "comportamento" e che, pertanto, non ridefinisca neppure `equals`.

```
public class R extends T {
  public R(int a) {
    super(a);
  }

  public void sayHi() {
    System.out.println("Hi!");
  }
}
```

[sorgente]

Supponiamo ora di sviluppare una classe di utilità con un metodo `isSmall` che restituisca `true` per tutti gli oggetti per cui `a` è `1`, oppure `2` e supponiamo (sebbene questo sia estremamente innaturale e inefficiente) di implementare tale metodo verificando l'appartenenza a un insieme `SMALLS` che contiene gli unici due oggetti che soddisfano tale requisito.

```
public class Client {
  private static final Set<T> SMALLS = Set.of(new T(1), new T(2));

  public static boolean isSmall(T t) {
    return SMALLS.contains(t);
  }
}
```

[sorgente]

La seguente porzione di codice mostra che il confronto tra oggetti di tipo diverso (anche se uno è sottotipo dell'altro) restituisce `false` (il che risolve i problemi di simmetria e transitività); mostra però anche che sostituendo il sottotipo `R` a `T` (vedi il codice evidenziato) il comportamento cambia!

```
public static void main(String[] args) {
S s = new S(1, 2);
T t = new T(1);
System.out.println(t.equals(s));
System.out.println(s.equals(t));

R r = new R(1);
System.out.println(Client.isSmall(t));
```

```
 System.out.println(Client.isSmall(r));
}
```

[sorgente]

L'output infatti è

```
false
false
true
false
```

Questo accade perché l'implementazione di `Set` si basa sul metodo `equals` per decidere se un oggetto appartiene all'insieme, ma gli elementi di `SMALLS` (che sono di tipo `T`) non risultano mai uguali a oggetti di tipo `R`!