Linguaggi di Programmazione 2

A.A. 2021-2022

Docente: M. Faella

Dispense tratte dal corso di Informatica a cura dello studente **S. Cerrone**

Sito Personale del Docente: wpage.unina.it/m.faella

Sommario

Qui ci sono degli esercizi che ho provato a svolgere: https://github.com/Tetrian/LdP2 (non vi aspettate nulla)

1. Introduzione

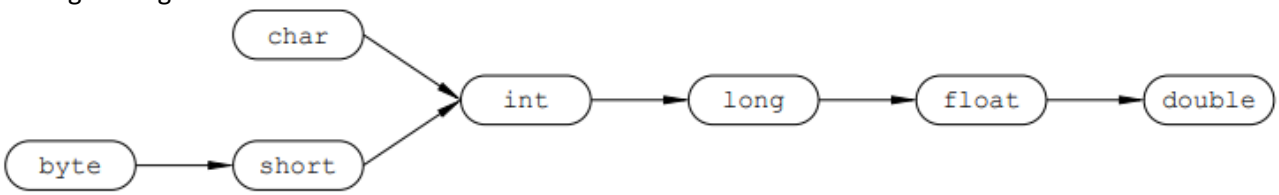
Tipi primitivi

In Java i tipi base sono otto: *boolean, char, byte, short, int, long, float e double*. A cui va aggiunto il tipo speciale *void* usato solo come tipo di ritorno dei metodi.

I tipi primitivi offrono una **conversione implicita** (o **promozione**) che non necessita di casting (il tipo $boolean$ non può essere mai convertito). Le conversioni possono avvenire in due modi:

- *Conversione di tipo per assegnazione*: avviene quando si assegna un valore ad una variabile di tipo differente e ciò può avvenire o con un operatore di assegnazione, o durante il passaggio di parametri ad un metodo
- *Promozione aritmetica*: in una espressione aritmetica, le variabili sono automaticamente promosse ad una forma più estesa per non causare perdita di informazioni.

Le conversioni legali (ovvero quelle che avvengono automaticamente) sono quelle per cui esiste un percorso nel seguente grafo:



Di norma, se è possibile la perdita di informazioni in una assegnazione o in un passaggio di parametri, il programmatore deve confermare l'assegnazione con un "cast" esplicito. In un cast esplicito il valore in eccesso viene troncato dalla cifra più significativa; dunque, bisogna assicurarsi che non ci siano perdite di informazioni (praticamente se si vuole percorrere il grafo al contrario bisogna farlo tramite cast espliciti).

Nota bene che è possibile una perdita di informazioni anche tramite promozioni, prendiamo ad esempio il seguente blocco di codice:

```
int i = 1000000001; // un miliardo e uno
float f = i; // qui f contiene un miliardo, perché un float non ha abbastanza bit
             // di mantissa per rappresentare le 10 cifre significative di i.
```

Se fosse stato solo un miliardo non avremmo avuto problemi, essendoci solo una cifra significativa. In ogni caso, sono possibili perdite di informazioni per le seguenti promozioni: $int \rightarrow float$, $int \rightarrow double$, $long \rightarrow float$, $long \rightarrow double$

Nota:

- float: 1 bit segno + 23 bit mantissa + 8 bit esponente = 32 bit (~7 cifre decimali significative)
- double: 1 bit segno + 52 bit mantissa + 11 bit esponente = 64 bit (~16 cifre decimali significative)

Aneddoto: Python ha un intero arbitrario (può usare più celle di memoria per rappresentare un singolo intero) e questo rende necessario avere una libreria per gestire questa cosa, e di conseguenza le operazioni sugli interi sono più lente.

Passaggio di argomenti

In Java, a differenza di altri linguaggi orientati agli oggetti (ad es., il C++), non esistono variabili che contengono oggetti, solo riferimenti ad oggetti, ovvero variabili che contengono l'indirizzo di un oggetto.

I riferimenti Java sono quindi simili ai puntatori del linguaggio C. Tuttavia, i riferimenti Java sono molto più restrittivi, infatti, non è possibile l'aritmetica dei puntatori (p++) o le conversioni tra puntatori e numeri interi (niente doppi puntatori, niente puntatori a funzioni). Quindi, rispetto ai puntatori, i riferimenti Java sono meno potenti, più facili da utilizzare e meno soggetti ad errori al run-time.

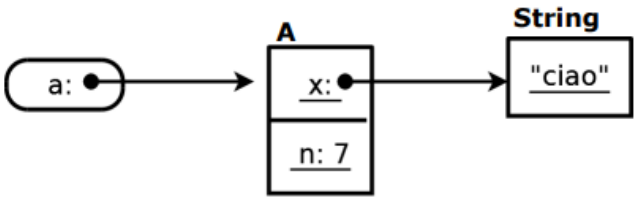
Java prevede solo il passaggio per copia: sia i tipi base che i riferimenti sono passati per copia. Non è possibile passare oggetti per valore, l'unico modo di manipolare (ed in particolare, passare) oggetti è tramite i loro riferimenti (ovvero, indirizzi).

Diagrammi di Memory Layout

Con la nozione di "memory layout" di un programma ci si riferisce ad una rappresentazione grafica dello stato della memoria ad un determinato punto di un programma (praticamente avrò un rettangolo per ogni blocco di memoria allocato a runtime).

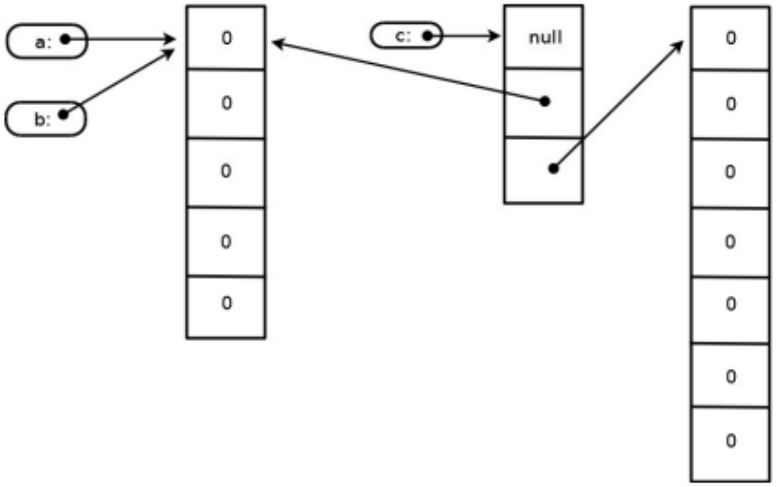
Come primo esempio (non evidenzieremo la differenza tra allocazione su stack e su heap), dato il frammento di codice Java avremmo il seguente memory layout:

```
class A {
    private String x;
    private int n;
    public A(String x, int n) {
        this.x = x;
        this.n = n;
    }
}
...
A a = new A("ciao", 7);
```



In Java, gli array sono oggetti a tutti gli effetti. In particolare, sono sottotipi di Object e ne ereditano i metodi. Di conseguenza non esistono variabili che contengono (direttamente) array, ma solo riferimenti ad array. Ad esempio, in figura viene riportato il memory layout del seguente frammento di programma:

```
int[] a = new int[5];
int[] b = a;
int[][] c = new int[3][];
c[1] = b;
c[2] = new int[7];
```



2. Costruttore ed Eccezioni

Concatenazione dei costruttori

Un costruttore può chiamare **esplicitamente** un altro costruttore della stessa classe usando la parola chiave ***this***, oppure il costruttore della superclasse usando la parola chiave ***super***. *this* e *super*, usati in questa accezione, devono comparire alla **prima riga** del costruttore, pena un errore di compilazione.

Ricordiamo che *this* e *super* hanno anche altri significati:

- *this* rappresenta il riferimento all'oggetto corrente
- *super* si usa per invocare un metodo di una superclasse, oppure in una classe interna per ottenere il riferimento all'oggetto che ha creato quello corrente.

Se un costruttore **non inizia** con una chiamata ad un altro costruttore (*this* o *super*), il compilatore fa una chiamata **implicita** al costruttore senza argomenti della superclasse, ovvero, inserisce automaticamente

l'istruzione *super()*. In tal caso, se la superclasse non ha un costruttore senza argomenti, si verifica un errore di compilazione.

Il meccanismo tramite il quale i costruttori possono invocarsi a vicenda prende il nome di **concatenazione dei costruttori** (constructor chaining).

Il compilatore controlla che la concatenazione **non sia ciclica**; infatti, se dei costruttori si chiamano a vicenda, siccome tali chiamate si trovano alla prima riga del rispettivo costruttore, e pertanto avvengono incondizionatamente, ci si trova in presenza di una mutua ricorsione non ben fondata, ovvero infinita.

Ad esempio, tendando di compilare la seguente classe:

```
class A {
    public A() { this(3); }
    public A(int i) { this(); }
}
```

si ottiene il seguente errore di compilazione:

```
A.java:3: recursive constructor invocation
    public A(int i) { this(); }
```

Per analizzare la concatenazione dei costruttori di una data gerarchia di classi, ed in particolare controllare che essa non sia ciclica, è possibile realizzare il seguente diagramma:

1) si crei un grafo con un nodo per ogni costruttore, compresi quelli impliciti
2) si aggiunga un arco da un nodo x ad un nodo y se il costruttore x chiama, esplicitamente o meno, y
3) il grafo ottenuto non deve presentare cicli

Eccezioni verificate e non verificate

Le classi di eccezione si dividono in **verificate** e **non verificate** (**checked** ed **unchecked**). Il termine "verificata" si riferisce al fatto che il compilatore verifica che tali eccezioni siano opportunatamente trattate dal programmatore, mentre le eccezioni non verificate sono ad uso libero. Intuitivamente, se in un metodo c'è un'istruzione che potrebbe lanciare un'eccezione verificata, il compilatore verifica che tale istruzione sia contenuta in un blocco *try...catch*, oppure che il metodo dichiari che può lanciare quella eccezione.

Se un metodo *foo* contiene l'istruzione *throw x*, oppure chiama un altro metodo la cui intestazione contiene la dichiarazione *throws x*, dove *x* è un'eccezione verificata, allora il compilatore controlla che sia rispettata una delle seguenti condizioni:

1) L'istruzione *throw* (oppure la chiamata) è contenuta in un blocco *try…catch* in cui una delle clausole *catch* è in grado di catturare *x*.
2) Il metodo *foo* contiene nell'intestazione la dichiarazione *throws y*, dove *y* è una superclasse di *x*

La seguente tabella definisce le combinazioni che sono corrette per il compilatore:

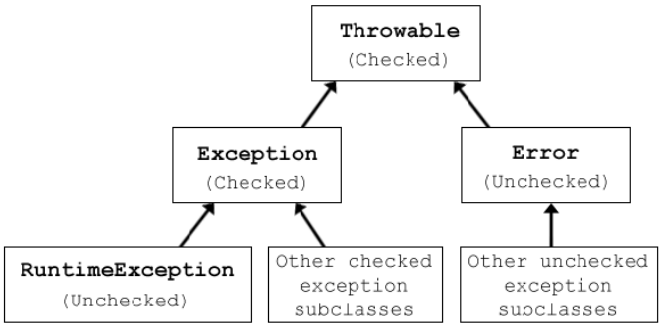
Come la gestisce il metodo \ Come nasce l'eccezione verificata x	***throw x***	**chiamando un metodo che "*throws x*"**
catch x	Assurdo!	sa gestire l'anomalia
throws x	segnala anomalia al chiamante	propaga l'anomalia verso il chiamante

- Delle quattro combinazioni che sono corrette per il compilatore, una non ha senso: che un metodo lanci direttamente un'eccezione con throw e subito la catturi con *try…catch*
- Le eccezioni servono per **segnalare una situazione anomala al chiamante**, mentre in questo caso l'eccezione sarebbe usata solo per modificare il flusso di controllo all'interno di un metodo, cosa per la quale esistono apposite istruzioni (in particolare, *if-then-else* e *break*)

In linea di massima, il criterio con cui scegliere se usare una eccezione di tipo checked o unchecked è il seguente: dovrebbero essere di tipo **verificato** quelle eccezioni che il programmatore **non può evitare** con certezza perché dipendono da **fattori esterni al programma**; viceversa, dovrebbero essere di tipo **non verificato** quelle eccezioni che derivano da **errori di programmazione** (e quindi evitabili).
Esempi:

- Si consideri l'eccezione che viene lanciata quando si cerca di aprire un file inesistente (*java.io.FileNotFoundException*). Per quanto si sforzi, il programmatore non può avere la certezza che tale eccezione non venga lanciata (anche se il programmatore controllasse l'esistenza del file subito prima di aprirlo, nulla vieta ad un altro programma di cancellare il file tra il controllo di esistenza e la sua apertura). Pertanto, l'eccezione *FileNotFoundException* è verificata.
- Si consideri l'eccezione che viene lanciata dalla JVM quando si tenta di accedere ad un array con un indice non valido (*ArrayOutOfBoundsException*). Questa eccezione è evitabile da parte del programmatore e difatti indica la presenza di un errore di programmazione; inoltre, se tale eccezione fosse verificata, il programmatore dovrebbe dichiararla o trattarla in occasione di ogni accesso ad array. Pertanto, *ArrayOutOfBoundsException* non è verificata.

Tutte le eccezioni della libreria standard sono **verificate** tranne quelle che discendono da *RuntimeException* e da *Error*. Quando si crea una nuova classe di eccezioni, tale classe eredita la categoria (checked oppure unchecked) dalla sua superclasse. Di seguito l'albero della gerarchia delle eccezioni:



3. Il sistema dei tipi e i tipi wrapper

Il sistema dei tipi

Il linguaggio Java è **staticamente** (il tipo di ogni espressione deve essere noto al momento della compilazione) **tipato** (ad ogni espressione viene assegnato un tipo, che rappresenta l'insieme di valori che l'espressione potrebbe assumere). In tal modo, il compilatore è in grado di controllare che tutte le operazioni siano applicate ad operandi compatibili (**type checking**).

In virtù del polimorfismo, bisogna distinguere due tipi di ogni variabile ed espressione di categoria "riferimento":

- Il tipo **dichiarato** (o statico) è noto a tempo di compilazione e rimane invariato durante tutta l'esecuzione
- Il tipo **effettivo** (o dinamico) non è noto a tempo di compilazione, nel senso che il compilatore non tenta di determinarlo, perché sarebbe in generale impossibile (si pensa a un parametro di un metodo). Il tipo effettivo può cambiare ogni volta che si assegna un nuovo valore a quel riferimento

Java prevede i seguenti tipi:

- I tipi base, o primitivi (introdotti nel capitolo 1)
- I tipi riferimento: sono quelli definiti dal nome di una classe, interfaccia o enumerazione
- I tipi array: sono tipi composti, ovvero si definiscono a partire da un altro tipo, detto tipo "componente", e si individuano sintatticamente per l'uso delle parentesi quadre
- Il tipo nullo: è un tipo speciale che prevede come univo valore possibile la costante *null*

Relazione di sottotipo

Per permettere il polimorfismo (in particolare, la capacità di un riferimento di puntare ad oggetti di tipo diverso), esiste una relazione binaria tra tipi, chiamata **relazione di sottotipo**, questa è definita dalle seguenti regole, in cui *T* ed *U* rappresentano tipi arbitrari (esclusi i tipi base):

1) *T* è sottotipo di sé stesso
2) *T* è sottotipo di *Object*
3) Se *T* estende *U* oppure implementa *U*, *T* è sottotipo di *U*
4) Il tipo nullo è sottotipo di *T*
5) Se *T* è sottotipo di *U* allora *T[]* è sottotipo di *U[]*

È evidente che la relazione di sottotipo è una relazione d'ordine sull'insieme dei tipi non base, gode infatti di proprietà riflessiva, antisimmetrica e transitiva. La relazione di sottotipo permette di definire precisamente il comportamento dell'operatore *istanceof*; infatti, data un'espressione *exp* ed il nome di una classe o interfaccia *T*, l'espressione *exp istanceof T* restituisce vero se e solo se il tipo **effettivo** di *exp* **non è nullo ed è sottotipo di *T*.**

Relazione di assegnabilità

La relazione di **compatibilità** (o **assegnabilità**) tra tipi stabilisce quando è possibile assegnare un valore di un certo tipo *T* ad una variabile di tipo *U*. Si dice che *T* è **assegnabile** ad *U* se *T* è sottotipo di *U*, oppure *T* ed *U* sono tipi base e c'è una conversione implicita da *T* ad *U*.

La relazione di assegnabilità si applica nei seguenti contesti:

- Assegnazione: *a = exp*
- Chiamata a metodo: *x.f(exp)*
- Ritorno da metodo: *return exp*

In base alla definizione di sottotipo, un array di qualunque tipo è sottotipo di "array di Object", questo consente, ad esempio, di passare qualunque array ad un metodo che abbia come parametro formale un array di Object. Consideriamo il seguente esempio:

```
String[] arr1 = new String[10];
Object[] arr2 = arr1;
arr2[0] = new Object();
String s = arr1[0];
```

L'esempio risulta corretto per il compilatore; tuttavia, nell'ultima istruzione assegnamo alla variabile *s*, dichiarata *String*, un oggetto di tipo effettivo *Object*, che non è compatibile. In effetti, al run-time viene sollevata un'eccezione (*ArrayStoreException*) al momento della **terza** istruzione perché al run-time gli array "ricordano" il tipo con il quale sono stati creati e la JVM utilizza questa informazione per controllare che gli oggetti inseriti nell'array siano sempre di tipo compatibile con quello dichiarato in origine.

I cast

Java permette alcune conversioni esplicite di tipo tramite cast con la sintassi *(T) exp*.
I cast tra tipi base sono sempre consentiti:

- Per effettuare esplicitamente una **promozione** (il cast è superfluo)
- Per effettuare una **promozione al contrario** (ad esempio da double a int): è facile incorrere in perdite di informazioni, per cui queste conversioni sono decisamente sconsigliate, al loro posto è opportuno utilizzare i metodi appositi della classe *Math*.

Sono consentiti dal compilatore i seguenti cast tra un tipo riferimento (o array) *A* ad un tipo riferimento (o array) *B* (negli altri casi, il cast porta a un errore di compilazione):

1) Se *B* è **supertipo** di *A* (upcast): è superfluo, perché i valori di tipo *A* sono di per sé assegnabili al tipo *B*

2) Se *B* è **sottotipo** di *A* (downcast): al run-time, la JVM controlla che l'oggetto da convertire appartenga effettivamente ad una sottoclasse di *B*, in caso contrario, viene sollevata l'eccezione *ClassCastException*. I downcast vanno evitati poiché aggirano il type checking svolto dal compilatore, se proprio si deve usare un downcast esso andrebbe preceduto da un controllo *istanceof* che assicuri la correttezza della conversione.

I tipi Wrapper

Per ogni tipo base, Java offre una corrispondente classe, che ingloba (wraps) un valore di quel tipo in un oggetto. Queste classi, dette **wrapper**, servono a trattare i valori base come se fossero oggetti. Le classi wrapper sono: *Byte, Short, Integer, Long, Float, Double, Boolean, Character, Void* (solo quest'ultimo non è un tipo base). Tutte le classi wrapper sono **immutabili** e **final** ed ogni classe wrapper ha un **costruttore** che accetta un valore del tipo base corrispondente. Esempio: *Integer n = new Integer(3);*

Ogni classe wrapper ha, inoltre, un metodo statico ***valueOf*** che prende come argomento un valore del tipo base corrispondente alla classe e restituisce un oggetto wrapper che lo ingloba. A differenza del costruttore, l'oggetto restituito non è necessariamente nuovo, ovvero, il metodo valueOf cerca di riciclare gli oggetti già creati (***caching***). In generale, questo non è un problema, perché gli oggetti wrappper sono immutabili.

Diamo una plausibile implementazione del metodo *valueOf*:

```
public static Integer valueOf(Int n) {
    private static final int CACHE_SIZE = 256; //da -127 a 128
    private static final Integer [] cache = new Integer [CACHE_SIZE]

    private static final int OFFSET = 127
    if (n >= -127 && n <= 128)
        return cache[n + OFFSET];
    else
        return new Integer(n);
}
// questo metodo presuppone che ci sia un blocco di codice che
// carica a load time il vettore, nella calsse Integer avrò quindi:
static {
    for (int i = 0, i < CACHE_SIZE, ++i)
        cache[i] = new Integer(i - OFFSET);
}
```

Queste variabili devono essere definite come attributi di classi e non all'interno del metodo come ho fatto io. Così è errore in compilazione

Diamo anche una seconda implementazione on-demand (che inserisce il numero quando serve):

```
1  public static Integer valueOf(Int n) {
2      private static final int CACHE_SIZE = 256; //da -127 a 128
3      private static final Integer [] cache = new Integer [CACHE_SIZE]
4
5      private static final int OFFSET = 127
6      if (n >= -127 && n <= 128) {
7          if (cache[n + OFFSET] != null)
8              return cache[n + OFFSET];
9          else
10             return cache[n + OFFSET] = new Integer(n);
11     }
12     else {
13         return new Integer(n);
14     }
15 }
```

Si noti che questo metodo non supporta il multithreading; infatti, è possibile che nelle linee 8 e 10 venga allocato due volte lo stesso intero se chiamato in contemporanea con due thread.

Le sei classi wrapper relative ai tipi numerici estendono la classe astratta *Number*, quest'ultima prevede sei metodi, che estraggono il valore contenuto, convertendolo nel tipo base desiderato:

```
public byte byteValue()
public short shortValue()
public int intValue()
public long longValue()
public float floatValue()
public double doubleValue()
```

Se un oggetto wrapper viene convertito in un valore base verso il quale non c'è una conversione implicita (ad esempio da double ad int), l'effetto sarà lo stesso di quello di un cast

Fino alla versione 1.4 di Java, era necessario convertire esplicitamente i valori dei tipi base in oggetti e viceversa, ma a partire dalla versione 1.5, questo procedimento è stato automatizzato introducendo l'**autoboxing** e l'**auto-unboxing**. Grazie a queste funzionalità, il compilatore si occupa di inserire le istruzioni di conversione laddove queste siano necessarie. Ad esempio, l'istruzione *Integer n = 7;* viene convertita in *Integer n = Integer.valueOf(7);* (e non in ~~*Integer n = new Integer(7);*~~).

Nota bene: essendo oggetti, per confrontare i tipi wrapper bisogna usare il metodo equals e non l'operatore ==; infatti applicando quest'ultimo avremmo un confronto tra riferimenti (come per tutti gli oggetti) e potremmo avere risultati inaspettati. Ad esempio:

```
Integer a = new Integer(7), b = new Integer(7);
System.out.println(a==b); // false, sono oggetti diversi

Integer a = 7, b = 7;
System.out.println(a==b); // true, perché viene chiamato il metodo statico valueOf che
                          // riutilizza gli oggetti corrispondenti a valori piccoli
Integer a = 700, b = 700;
System.out.println(a==b); // false, perché il metodo valueOf riutilizza soltanto
                          // gli interi compresi tra -127 e 127
```

4. Risoluzione dell'overloading e dell'overriding

Questo capitolo non è programma di quest'anno.

Binding dinamico

Per binding dinamico (bind significa legare) si intende il meccanismo per cui non è il compilatore, ma la JVM ad avere l'ultima parola su quale metodo invocare in corrispondenza di ciascuna chiamata a metodo. In effetti, questo meccanismo è una diretta conseguenza del polimorfismo e dell'overriding, ovvero, ciascun riferimento può puntare ad oggetti di tipo effettivo diverso (polimorfismo) e ciascuno di questi tipi effettivi può prevedere una versione diversa dello stesso metodo (overriding).

Inoltre, il compilatore non può prevedere di che tipo effettivo sarà una variabile nel corso dell'esecuzione del programma (tecnicamente, questo problema è indecidibile).

Quindi, in Java il binding (collegare ciascuna chiamata ad un metodo vero e proprio) avviene in due fasi:

- **Early binding**, in cui il compilatore risolve l'overloading (scegliendo la firma più appropriata alla chiamata)
- **Late binding**, in cui la JVM risolve l'overriding (scegliendo il metodo vero e proprio).

È chiaro che il late binding non è necessario per i metodi **privati**, **statici** o **final** che non ammettono overriding. Infatti, per questi metodi si parla di **binding statico**, perché la scelta del metodo da eseguire viene fatta già dal compilatore.

Fasi del binding

L'early binding si divide a sua volta in due fasi:

1) Individuazione delle firme candidate
    - Consideriamo una generica invocazione $x.f(a_1, \ldots, a_n)$

- Una generica firma $f(T_1, \ldots, T_n)$ (per firma di un metodo si intende il suo nome e l'elenco dei tipi dei suoi parametri formali) è **candidata** per la chiamata in questione se:
  - Si trova nella classe dichiarata di *x* o in una sua superclasse
  - È **visibile** dal punto della chiamata, rispetto alle regole di visibilità di Java
  - È **compatibile** con la chiamata; ovvero, per ogni indice *i* compreso tra 1 ed n, il tipo (dichiarato) del parametro attuale a_i è assegnabile al tipo T_i
- Se nessuna firma risulta candidata per una data chiamata, il compilatore segnala un errore (accompagnato dal messaggio: *"cannot find symbol"*).

2) Scelta della firma più specifica tra quelle candidate
   - Confrontare le firme candidate:
     - Date due firme con lo stesso nome e numero di argomenti $f(T_1, \ldots, T_n)$ e $f(U_1, \ldots, U_n)$
     - Si dice che la prima firma è **più specifica** della seconda se, per ogni indice *i* compreso tra 1 ed n, il tipo T_i è assegnabile al tipo U_i (nota bene che questo confronto tra firme non dipende dal tipo dei parametri attuali passati alla chiamata)
     - È facile verificare che essere "più specifico" è una relazione d'ordine (riflessiva, antisimmetrica e transitiva) sull'insieme delle firme. Tale ordine è **parziale**, in quanto alcune firme non sono confrontabili tra loro. Ad esempio, le firme *f(int, double)* e *f(double, int)* non sono confrontabili quanto a specificità.
   - L'early binding si conclude individuando, tra le firme candidate, una che sia **più specifica di tutte le altre**
     - Per simulare "a mano" questo meccanismo, nei casi complessi può essere conveniente realizzare un diagramma, in cui ci sia un nodo per ciascuna firma candidata ed un arco orientato da un nodo a ad un nodo b quando la firma a è più specifica della firma b
     - Se nel diagramma c'è un nodo che ha archi uscenti diretti verso tutte le altre firme, quella sarà la firma scelta dal compilatore
     - Se nessuna firma è più specifica di tutte le altre, il compilatore segnala un errore e termina (si parla in questo caso di **chiamata ambigua**)

Il late binding è la fase di risoluzione dell'**overriding**, a carico della JVM. Questa fase riceve in input la firma scelta dal compilatore durante l'early binding.

- Consideriamo nuovamente la generica invocazione $x.f(a_1, \ldots, a_n)$
  La JVM cerca un metodo da eseguire, con il seguente algoritmo:
  - si parte dalla classe effettiva di *x*
  - si cerca un metodo che abbia la firma identica a quella scelta dall'early binding
  - se non lo si trova, si passa alla superclasse
  - così via, fino ad arrivare ad Object
- Questo procedimento può fallire, cioè non trovare alcun metodo, solo in casi molto particolari
  - Ad esempio, se una classe *A* dipendeva da una classe *B* e la classe *B* è cambiata da quando è stata compilata *A*

Early binding e autoboxing

Nella risoluzione dell'overloading, l'autoboxing e l'auto-unboxing entrano in gioco soltanto se necessario, ovvero, soltanto se altrimenti non ci sarebbero firme candidate. Quindi, come primo tentativo, il compilatore cerca le firme che sono candidate senza prendere in considerazione l'autoboxing e l'auto-unboxing, **solo se non ci sono firme candidate**, il compilatore abilita le conversioni da tipo primitivo a tipo wrapper, e viceversa, e riesamina tutte le firme (secondo tentativo).

Questa scelta è stata fatta per mantenere la compatibilità con il codice scritto prima dell'introduzione dell'auto-(un)boxing. Infatti, le invocazioni a metodo che funzionavano senza auto-(un)boxing continuano a funzionare con l'auto-(un)boxing, e sono risolte nello stesso modo. Mentre, con l'auto-(un)boxing, alcune invocazioni che prima non erano consentite diventano lecite.

Late binding e autoboxing: Una volta ottenuto un insieme non vuoto di firme candidate, il compilatore passa alla scelta della più specifica, con le regole descritte precedentemente. Quindi, l'auto-(un)boxing **non influenza** in alcun modo la scelta della firma più specifica. Analogamente, l'auto-(un)boxing **non influenza** in alcun modo il **late binding**.

Early binding e autoboxing: esempi

1) Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }
public static int foo(long i, String o) { return 2; }
```

- La chiamata *foo(new Integer(7), "ciao")* provoca un errore di **ambiguità**, perché il compilatore prima ottiene un insieme di firme candidate vuoto; poi, una volta attivato l'auto-(un)boxing, ottiene candidate **entrambe** le firme *foo*, delle quali nessuna è più specifica dell'altra
- La chiamata *foo(new Float(7), "ciao")* provoca un **errore** di compilazione, in quanto il compilatore non trova firme candidate neanche al secondo tentativo
- La chiamata *foo(new Long(7), "ciao")*ottiene output ***2***, in quanto quella è l'unica firma candidata, una volta attivato l'auto-(un)boxing

2) Consideriamo i seguenti metodi:

```
public static int bar(double a, Integer b) { return 3; }
public static int bar(Double a, Integer b) { return 4; }
```

- La chiamata *bar(1.0, 7)* provoca un errore di ambiguità, perché entrambe le firme saranno candidate (al secondo tentativo)
- La chiamata *bar(1, 7)* ottiene il risultato *3*, perché avrà un'unica firma candidata (al secondo tentativo). La seconda firma non sarà, infatti, candidata perché un *int* non può trasformarsi in *Double* tramite autoboxing.

5. Controllo di uguaglianza tra oggetti

Implementazione tipica

Nel linguaggio Java, l'operatore == stabilisce se due riferimenti puntano al medesimo oggetto. Spesso, è utile considerare uguali due oggetti distinti, secondo un criterio dettato di volta in volta dal contesto applicativo.

Il modo standard di confrontare oggetti è tramite il metodo ***equals***, definito dalla classe *Object*:

```
public boolean equals(Object x)
```

In generale, il metodo *equals* prende come argomento un oggetto *x* e restituisce vero se questo oggetto (*this*) è da considerarsi "uguale" ad *x*, e falso altrimenti (nella classe *Object* si comporta come ==).

Ad esempio, consideriamo un programma di gestione del personale dove la classe *Employee* rappresenta un impiegato ed è caratterizzata dal seguente schema:

```
class Employee {
    private String name;    // nome impiegato
    private int salary;     // salario mensile
    private Employee boss;  // capoufficio (altro impiegato)
    ...
}
```

Supponiamo che due impiegati vadano considerati uguali se hanno lo stesso nome e lo stesso capoufficio (l'applicazione suppone che non ci possano essere due impiegati omonimi all'interno dello stesso ufficio) e che la relazione "essere capoufficio di" disponga gli impiegati in un albero, alla cui radice vi è un impiegato senza capoufficio (*boss == null*).

Una possibile implementazione del metodo equals per la classe *Employee* è la seguente:

```
public boolean equals(Object o) {
    if (o == null) return false; // questo controllo è ridondante
    if (!(o instanceof Employee)) return false;
    Employee e = (Employee) o;
    return name.equals(e.name) &&
        (boss == e.boss || (boss!=null && boss.equals(e.boss)));
}
```

Si ricorda che instanceof restituisce false se *o* è null

- All'inizio, controlliamo che il riferimento passato punti effettivamente ad un Employee
- Confrontiamo i nomi con equals
- Confrontiamo anche i capiufficio con equals (sono impiegati anche loro, quindi vale per loro la stessa assunzione fatta per gli impiegati semplici)
- La forma complessa dell'espressione condizionale è dovuta al caso in cui uno dei due impiegati in questione, o entrambi, siano al vertice dell'azienda

Non c'è un metodo corretto per definire l'uguaglianza tra classi (nell'esempio precedente poteva bastare semplicemente il nome, ad esempio) ma il linguaggio Java richiede che **qualunque** ridefinizione del metodo equals rispetti le seguenti proprietà (condizioni proprie di una relazione di equivalenza):

- **Riflessività**: per ogni oggetto *x*, *x.equals(x)* è vero
- **Simmetria**: per ogni coppia di oggetti *x* ed *y*, *x.equals(y)* è vero se e solo se *y.equals(x)* è vero
- **Transitività**: per ogni terna di oggetti *x, y* e *z*, se *x.equals(y)* è vero e *y.equals(z)* è vero, allora anche *x.equals(z)* è vero

Metodo equals ed ereditarietà

Bisogna prestare attenzione all'interazione tra il metodo equals e le **sottoclassi**, ad esempio, supponiamo che la classe *Employee* abbia una sottoclasse ***Manager*** con un campo aggiuntivo intero chiamato "bonus". In base al contesto applicativo, in fase di progettazione, va deciso come si deve comportare il metodo equals con oggetti appartenenti a sottoclassi diverse, in particolare bisogna porsi le seguenti domande:

1) Il criterio di confronto tra oggetti di una sottoclasse (ad esempio, due *Manager*) è diverso da quello che vale tra oggetti della superclasse (ad esempio due *Employee*)?
2) Un oggetto di una sottoclasse (esempio, *Manager*) può essere considerato uguale ad uno di una superclasse (esempio *Employee*)?

In pratica ci sono due scenari standard che, oltre a rispondere ai quesiti precedenti, coprono il 90% delle specifiche reali:

1) Il criterio di confronto è **uniforme** in tutta la gerarchia e coincide con quello che si applica alla sua radice; quindi, oggetti di sottoclassi diverse possono anche risultare uguali tra loro
2) Il criterio di confronto **cambia** nelle sottoclassi; ovvero, oggetti di sottoclassi diverse sono sempre considerati diversi

Criterio uniforme

Nel primo scenario, in tutta la gerarchia vale il criterio stabilito per la sua radice, ad esempio, supponiamo che la classe Employee abbia un campo "codice fiscale" che identifica univocamente una persona. Quindi, è ragionevole confrontare soltanto quel campo per stabilire se due *Employee* sono uguali; inoltre, lo stesso criterio si può applicare anche a due *Manager*, oppure per confrontare un *Employee* e un *Manager*.

L'implementazione di questo scenario è estremamente semplice:

- La radice della gerarchia (ad esempio, *Employee*) fornisce un'implementazione di equals
- Questa versione è etichettata con ***final***, in modo da non poter essere sovrascritta nelle sottoclassi
- Le sottoclassi non contengono ulteriori versioni di equals

Criterio non uniforme

Il criterio di confronto cambia nelle sottoclassi, solitamente, vuol dire che ciascuna classe controlla che i propri campi siano uguali, nei due oggetti da confrontare. In questo scenario, il metodo equals va opportunamente **ridefinito in ogni sottoclasse** che abbia un proprio criterio di uguaglianza.

Ad esempio, la seguente specifica rientra nel secondo scenario:

- due *Employee* sono uguali se hanno lo stesso nome e salario
- due *Manager* devono avere anche lo stesso bonus
- un *Employee* non è mai uguale ad un *Manager*

In generale, il metodo equals delle sottoclassi dovrebbe **raffinare** la relazione di uguaglianza, cioè renderla più selettiva, e non viceversa.

Nell'implementazione del secondo scenario, bisogna fare in modo che ciascuna classe si occupi solo dei propri campi, e deleghi alle sue superclassi il confronto dei campi ereditati; quindi, il metodo equals nelle sottoclassi (ad esempio *Manager*) dovrebbe preliminarmente invocare quello della superclasse:

```
if (!super.equals(other)) return false;
```

Poi, se l'oggetto ricevuto come argomento è anch'esso un *Manager*, dovrebbe procedere ad effettuare i confronti specifici dei *Manager*.

L'obiettivo è confrontare solo oggetti dello **stesso tipo effettivo**, poiché in questo scenario, oggetti di tipo effettivo diverso sono sempre considerati diversi. Ma non è possibile effettuare questo genere di controllo con l'operatore instanceof perché esso tratta allo stesso modo una classe e tutte le sue sottoclassi; dunque, è necessario usare la classe **Class** e il metodo ***getClasss*** della classe Object.

Un esempio di test è il seguente: `if (o.getClass() != getClass()) return false;`

In questo modo, sono scartati come differenti soltanto gli oggetti di tipo effettivo diverso, come previsto dallo scenario e come ulteriore beneficio, nel metodo equals della classe *Manager*, ad esempio, supposto che quel test sia fatto in *Employee*, non sarà necessario verificare che l'oggetto ricevuto come argomento sia un *Manager*, perché questo test sarà già effettuato da equals di *Employee*.

Dunque, nel caso in cui adottiamo il secondo scenario standard, otteniamo una soluzione di questo tipo:

```
// In Employee:
public boolean equals(Object o) {
    if (o==null) return false; // controllo non ridontante
    if (o.getClass()!=getClass()) return false;
    Employee e = (Employee) o; // confronta i campi di Employee
    ...
}
// In Manager:
public boolean equals(Object o) {
    if (!super.equals(o)) return false;
    Manager m = (Manager) o; // confronta i campi di Manager
    ...
}
```

Confronti misti

I due scenari standard non coprono tutti i casi possibili, in particolare, sono esclusi i casi in cui il criterio non è uniforme, e oggetti di classi diverse possono in alcune circostanze essere considerati uguali.

Come esempio concreto possiamo considerare la seguente specifica:

- due *Employee* sono uguali se "hanno lo stesso nome e salario"
- due *Manager* sono uguali se "hanno lo stesso nome, salario e bonus"
- un *Employee* e un *Manager* sono uguali se "hanno lo stesso nome e salario, e il Manager ha il bonus pari a zero"

Sorprendentemente, l'implementazione di questa specifica è estremamente complessa. Almeno se si vogliono rispettare i principi per il quale le classi non dovrebbero essere consapevoli delle loro sottoclassi e ciascuna classe dovrebbe controllare solo i propri campi (specifiche di questo tipo sono **sconsigliate**). Diamo comunque un esempio di implementazione (corretta, ma poco pulita e leggibile):

```
// In Employee:
public boolean equals(Object o) {
    if (!(o instanceof Employee)) return false;
    Employee e = (Employee) o;
    if (!(name.equals(e.name) && salary==e.salary)) return false;
    if (getClass()== Employee.class) return e.isCompatibleWithEmployee();
    if (e.getClass()== Employee.class) return isCompatibleWithEmployee();
    return true;
}
public boolean isCompatibleWithEmployee() { return true; }

// In Manager:
public boolean equals(Object o) {
    if (!super.equals(o)) return false;
    if (!(o instanceof Manager)) return true;
    Manager m = (Manager) o;
    return bonus == m.bonus;
}
public boolean isCompatibleWithEmployee() { return bonus == 0; }
```

6. Nesting

Classi interne

Java permette di definire una classe (o interfaccia) all'interno di un'altra, queste classi vengono chiamate **interne** (o **annidate**, in inglese nested). In particolare, le classi interne non statiche sono chiamate ***inner***.

Tale meccanismo arricchisce le possibilità di relazioni tra classi, introducendo in particolare **nuove regole di visibilità**.

Le classi interne (non statiche) godono delle seguenti proprietà distintive:

1) **Privilegi di visibilità** rispetto alla classe contenitrice e alle altre classi in essa contenute: permettono una stretta collaborazione tra queste classi
   - Dall'esterno di *A*, i nomi completi delle classi *B* e *C* sono *A.B* e *A.C*, rispettivamente
   - La visibilità di una classe interna non ha alcun effetto sul codice che si trova all'interno della classe che la contiene. Ad esempio, la classe *B* dell'immagine a destra è visibile a tutto il codice contenuto in *A*, compreso il codice contenuto in altre classi interne ad *A*, come ad esempio *C*

```
public class A {
    private class B {
        ...
    }
    class C {
        ...
    }
}
```

- Lo stesso discorso si applica per i campi e i metodi di una classe interna. I loro attributi di visibilità hanno effetto solo sul codice **esterno** alla classe contenitrice
- In altre parole, **tra classi contenute nella stessa classe non vige alcuna restrizione di visibilità**

2) **Restrizioni di visibilità** rispetto alle classi esterne a quella contenitrice: permettono di nascondere la classe all'esterno (incapsulamento)
- Oltre a campi e metodi, una classe può contenere altre classi o interface, dette **interne**. Una classe che non sia interna viene chiamata **top-level**
- A differenza delle classi top-level, le classi interne possono avere **tutte le quattro visibilità** ammesse dal linguaggio
- La visibilità di una classe interna *X* stabilisce quali classi possono utilizzarla (cioè, istanziarla, estenderla, dichiarare riferimenti o parametri di tipo *X,* etc…)
- Riprendendo l'esempio precedente, la classe *B* **non è visibile al di fuori di *A*,** mentre la classe *C* è visibile a tutte le classi nello stesso pacchetto di *A*

3) Un **riferimento implicito** ad un oggetto della classe contenitrice: ogni oggetto della classe interna "conosce" l'oggetto della classe contenitrice che l'ha creato.
- Ciascun oggetto di una classe interna (non statica) possiede un riferimento implicito ad un oggetto della classe contenitrice.
- Tale riferimento viene inizializzato automaticamente al momento della creazione dell'oggetto e non può essere modificato
- Supponiamo che *B* sia una classe interna di *A*. Se viene creato un oggetto di tipo *B* in un contesto non statico della classe *A*, il riferimento implicito verrà inizializzato con il valore corrente di *this*. In tutti gli altri casi, è necessario utilizzare una nuova forma dell'operatore *new*, ovvero:
```
<riferimento ad oggetto di tipo A>.new B(...)
```
- All'interno della classe *B*, la sintassi per denotare questo riferimento implicito è *A.this*. L'uso di *A.this* è facoltativo, come quello di *this*, cioè, si può accedere ad un campo o metodo *f* della classe *A* sia con *A.f* che con *f*
- Nota: una classe interna non statica non può avere membri (campi o metodi) statici

Come già accennato le classi interne possono essere statiche o meno, una classe interna dichiarata nello scope di classe (cioè al di fuori di metodi e iniizializzatori) è statica se preceduta dal modificatore ***static***. Una classe interna dichiarata all'interno di un metodo (quindi, locale) o di un inizializzatore eredita il proprio essere statica o meno dal metodo in cui è contenuta o dal campo che si sta inizializzando.

Le classi **interne statiche non possiedono il riferimento implicito** alla classe contenitrice. Quindi, le classi interne statiche godono solo delle prime due proprietà descritte precedentemente.

Classi locali

Una classe interna dichiarata all'interno di un metodo viene chiamata **locale**. Una classe locale non ha specificatore di visibilità, in quanto è visibile **solo** all'interno del metodo in cui è dichiarata. Inoltre; una classe locale non può avere il modificatore *static*, in quanto eredita il suo essere statica o meno dal metodo in cui è dichiarata.

Oltre a godere delle proprietà comuni alle classi interne, le classi locali "vedono" le variabili locali e i parametri formali del metodo in cui sono contenute, a patto che essi siano **effectively final** (vedi dopo).

Le istanze di una classe locale possono vivere più a lungo del metodo in cui la loro classe è visibile; tipicamente, questo succede quando un oggetto di una classe locale viene restituito dal metodo, mascherato da una superclasse o super-interfaccia nota all'esterno.
In questi casi, l'accesso alle variabili locali (compresi i parametri formali) di quel metodo può avvenire quando quel metodo è ormai terminato, cancellando di fatto quelle variabili (si ricordi che le variabili locali e i parametri formali sono allocati sullo stack e quindi vengono cancellati al termine di ciascuna invocazione

nel metodo). Pertanto, per dare l'illusione al programmatore di accedere a quelle variabili, il compilatore in realtà inserisce negli oggetti delle classi locali delle copie di quelle variabili.

Se il meccanismo funzionasse anche per variabili non final, il "trucco" delle copie delle variabili locali diventerebbe visibile al programmatore, invece che nascosto. Infatti, potrebbe accadere che una variabile locale venisse modificata dopo la creazione di un oggetto della classe interna, questo, interrogato successivamente, ricorderebbe il vecchio valore di quella variabile, invece del valore modificato.

Diamo un esempio usando l'interfaccia *ActionListener* della libreria AWT (Abstract Windowing Toolkit) che rappresenta un oggetto in grado di rispondere ad un evento:

```
interface ActionListener { void actionPerformed(ActionEvent e); }
```

Il seguente frammento, invece, crea due widget grafici:

```
final JLabel label = new JLabel("Hello"); // scritta non editabile
JButton b = new JButton(...); // pulsante
```

Alla pressione del pulsante, vogliamo modificare il contenuto della scritta. Usiamo una classe locale per creare un oggetto che implementi *ActionListener*:

```
class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // reagisci alla pressione del tasto b
        label.setText("Goodbye");
    }
}
b.addActionListener(new MyActionListener());
```

Classi anonime

Una classe locale può anche essere **anonima**, quando il suo nome non sarebbe rilevante e/o utile. Talvolta una classe interna viene utilizzata soltanto una volta, tipicamente per istanziare un oggetto che poi viene mascherato con una classe o interfaccia esistente; in questi casi, si può utilizzare una classe anonima.

Una **classe anonima** si definisce a partire da una classe o una interfaccia esistente, nella seguente tabella riportiamo a sinistra la sintassi per la creazione di classi anonime a partire da una classe o interfaccia ed a destra il costrutto equivalente utilizzando normali classi con nome:

`new Classe(exp_1, ..., exp_n) {`
`  <body>`
`}`
	`class X extends Classe {`
	`  public X(T_1 x_1, ..., T_n x_n) {`
	`  super(x_1, ..., x_n);`
	`  }`
	`  <body>`
	`}`
	`new X(exp_1, ..., exp_n)`

`new Interfaccia() {`
`  <body>`
`}`
	`class X implements Interfaccia {`
	`  <body>`
	`}`
	`new X()`

Esempio di classe anonima da interfaccia:

- Nel seguente frammento viene usata una classe anonima per creare un oggetto che implementi *ActionListener*

```
Jbutton b = new JButton(...);
b.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // regisci alla pressione del tasto b
    }
});
```

Esempio di classe anonima da classe:

- Consideriamo la solita classe *Employee*, dotata di campi nome e salario
- Il seguente metodo usa una classe anonima per creare un *Employee* dotato di un titolo (Dott., Prof., etc…) che comparirà nella stringa restituita dal metodo *toString*

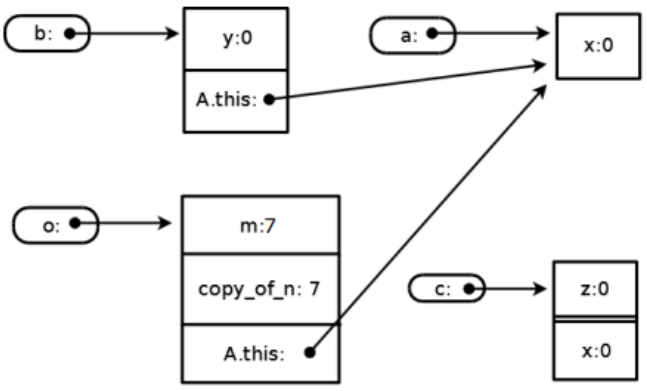
```
public static Employee makeSpecialEmployee
                        (String title, String name, int salary) {
    return new Employee(name, salary) {
        @Override
        public String toString() {
        return title + " " + getName() + ": " + getSalary();
        }
    };
}
```

- Il parametro formale "*title*" è visibile all'interno della classe anonima in quanto **effectively final**

Memory layout di classi interne

La figura rappresenta il memory layout delle classi definite in seguito (si noti in particolare la differenza tra classe interna e sottoclasse)

```
class A {
    private int x;
    private class B {
        private int y;
    }
    public static class C extends A {
        private int z;
    }
    public Object f(final int n) {
        class Locale {
            private int m = n;
        }
        return new Locale();
    }
    public static void main(String[] args) {
        A a = new A();
        B b = a.new B();
        C c = new C();
        Object o = a.f(7);
    }
}
```



7. Generics e Iterator

Classi parametriche

La programmazione parametrica, detta anche **generics**, è una importante funzionalità nel linguaggio che consente di dotare classi, interfacce e metodi di parametri di tipo; quest'ultimi sono simili ai normali parametri dei metodi ma hanno come possibili valori tutti i tipi (non primitivi) del linguaggio.

Il meccanismo dei generics consente di scrivere codice più robusto dal punto di vista dei tipi di dati, evitando in molti casi il ricorso alle conversioni forzate (cast). Questa funzionalità è una forma di **polimorfismo parametrico**.

La programmazione parametrica dimostra tutta la sua utilità nelle realizzazioni di collezioni (classi che contengono altri oggetti); quindi, come primo esempio supponiamo di voler realizzare una classe, chiamata *Pair*, che rappresenta una coppia di oggetti dello stesso tipo.

In mancanza della programmazione parametrica la classe si sarebbe dovuta realizzare secondo il seguente schema (o peggio creare una classe *Pair* per ogni tipo):

```
class Pair {
    private Object first, second;
    public Pair(Object a, Object b) { ... }
    public Object getFirst() { ... }
    public void setFirst(Object a) { ... }
    ...
}
```

Ma, un'implementazione come questa comporta che gli utenti della classe debbano necessariamente ricorrere al **cast** così che gli elementi estratti dalla coppia riacquistino il loro tipo originario, come ad esempio `Pair p = new Pair("uno", "due");  String a = (String) p.getFirst();`

Per ovviare questo problema basta rendere la classe *Pair* parametrica nel seguente modo:

```
class Pair<T> {
    private T first, second;
    public Pair(T a, T b) {
        first = a;
        second = b;
    }
    public T getFirst() { return first; }
    public void setFirst(T a) { first = a; }
    ...
}
```

- In questo caso, la classe *Pair* ha un parametro di tipo, chiamato *T*
- I parametri di tipo vanno dichiarati dopo il nome della classe, racchiusi tra parentesi angolari
- Se vengono dichiarati più parametri di tipo, questi vanno separati da virgole
- All'interno della classe, un parametro di tipo si comporta **come un tipo di dato vero e proprio**, tranne che per alcune eccezioni che vedremo in seguito
- In particolare, come si vede dall'esempio, un parametro di tipo si può usare come tipo di un campo, tipo di un parametro formale di un metodo e tipo di ritorno di un metodo.

La nuova versione di *Pair* permette agli utenti della classe di specificare di che tipo di coppia si tratta evitando il cast: `Pair<String> p = new Pair<String>("x","y"); String a = p.getFirst();`

- Nella dichiarazione della variabile *p*, è **obbligatorio** indicare il parametro attuale di tipo
- Nell'instanziazione dell'oggetto *Pair*, tale indicazione è **facoltativa** (in tale contesto)

```
Pair<String> p = new Pair<>("uno", "due");
String a = p.getFirst();
```

```
Pair<Employee> q = new Pair<>(…, …);
Employee e = q.getFirst();
```

Come per i normali parametri dei metodi, *String* è il **parametro attuale**, che prende il posto del **parametro formale** "T" di *Pair*.

Esaminiamo anche un'ulteriore versione di *Pair*, in grado di contenere due oggetti di tipo diverso; in questo caso, la classe avrà due parametri di tipo, che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia:

```
class Pair<T, U> {
    private T first;
    private U second;
    public Pair(T a, U b) {
        first = a;
        second = b;
    }
    public T getFirst() { return first; }
    public void setFirst(T a) { first = a; }
    public U getSecond() { return second; }
    public void setSecond(U a) { second = a; }
}
Pair<String,Employee> p = new Pair<>("Pippo", new Employee(…));
```

La versione grezza delle classi parametriche

Per compatibilità con le versioni precedenti di Java, è possibile usare una classe parametrica come se non lo fosse. Quando utilizziamo una classe parametrica senza specificare i parametri di tipo, si dice che stiamo usando la versione **grezza** di quella classe.

Questa possibilità è stata data per retrocompatibilità, infatti dopo l'introduzione dei generics, molte classi della libreria standard Java sono diventate parametriche ma la versione grezza di queste classi permette alla nuova versione della libreria standard di essere compatibile con i programmi scritti con le versioni precedenti del linguaggio.

Ad esempio, se *Pair* è la classe parametrica descritta precedentemente, è anche possibile utilizzarla così:

```
Pair p = new Pair("uno", "due");     // Provoca un warning
String a = (String) p.getFirst();    // cast indispensabile
```

N.B.: Le classi grezze esistono solo per retrocompatibilità, il codice nuovo **dovrebbe sempre specificare** i parametri di tipo delle classi parametriche.

Metodi parametrici

Una classe è parametrica se ha almeno un parametro di tipo, anche i singoli metodi e costruttori possono avere parametri di tipo, indipendentemente dal fatto che la classe cui appartengono sia parametrica o meno. Un caso tipico è rappresentato dai metodi statici, i quali non possono utilizzare i parametri di tipo della classe in cui sono contenuti.

Il seguente metodo parametrico restituisce l'elemento mediano (di posto intermedio) di un dato array:

```
public static <T> T getMedian(T[] a) {
    int l = a.length;
    return a[l/2];
}
```

- Come si vede, il parametro di tipo va dichiarato **prima del tipo restituito**, racchiuso tra parentesi angolari, questo parametro è **visibile solo all'interno del metodo**
- In questo caso, il parametro di tipo permette di restituire un oggetto dello stesso tipo dell'array ricevuto come argomento

Quando si invoca un metodo parametrico, è opportuno, ma non obbligatorio, specificare il parametro di tipo attuale per quella chiamata, ad esempio, supponendo che il metodo *getMedian* precedentemente descritto appartenga ad una classe *Test*, lo si può invocare con *String s = test.<String>getMedian(x);*
Il parametro attuale di tipo va quindi indicato tra il punto e il nome del metodo, ma è anche possibile ommetterlo; in questo caso, il compilatore cercherà di **dedurre** il tipo più appropriato, mediante un meccanismo chiamato **type inference** (inferenza di tipo).

Anche i costruttori possono essere parametrici, indipendentemente dal fatto che la loro classe sia parametrica o meno:

```
public class A<T> {
    public <U> A(T x, U y) {...}
}
```

In quest'esempio, il costruttore della classe parametrica *A* ha a sua volta un parametro di tipo chiamato *U*, mentre il parametro *T* è visibile in tutta la classe *A*, il parametro *U* è visibile solo all'interno di quel costruttore. Il costruttore in questione può essere invocato con la seguente sintassi:

```
A<String> a = new <Integer>A<String>("ciao", new Integer(100));
```

- Il parametro di tipo del costruttore (Integer) va specificato prima del nome della classe
- Il parametro di tipo della classe, come già visto per *Pair*, va specificato dopo il nome della classe

Type inference

Per sommi capi, la type inference cerca di individuare il tipo più specifico che rende la chiamata corretta. L'algoritmo di type inference non è né corretto né completo; infatti, può fallire anche quando una soluzione esiste oppure può individuare una soluzione anche quando questa non esiste; in questo caso, la soluzione individuata sarà segnalata successivamente come errore dal type checking.

Le regole precise che il compilatore adotta nella type inference non saranno trattate, ci limiteremo ad esaminare alcuni esempi per capirne il meccanismo.

Con riferimento al metodo *getMedian*, è possibile invocarlo senza specificare il parametro attuale di tipo:

```
String[] x = {"uno", "due", "tre"};     String s = Test.getMedian(x);
```

Il compilatore **dedurrà correttamente** che il tipo desiderato è String grazie alla type inference.

Consideriamo invece il seguente metodo, che assegna il medesimo riferimento a tutte le celle di un array:

```
public static <T> void fill(T[] a, T x) {
    for (int i = 0; i < a.length; ++i)
        a[i] = x;
}
```

L'intenzione del programmatore è chiaramente quella di accettare un array e un oggetto dello stesso tipo, come ad esempio `String[] a = new String[10];  fill(a, "ciao");` in questo caso, il compilatore, durante la fase di type inference, dedurrà correttamente che il tipo desiderato è String.

Con riferimento allo stesso metodo *fill*, se consideriamo invece il seguente frammento:

```
Employee[] a = new Employee[10];   fill(a, new Integer(100));
```

Siccome l'array e l'oggetto passato sono di due tipi differenti, ci aspettiamo che la type inference **fallisca**, producendo un errore di compilazione. Invece, la compilazione **va a buon fine**, mentre l'esecuzione del programma produce il lancio di un'**eccezione:**

- la compilazione termina con successo perché la type inference deduce che il tipo richiesto è *Object*
- in base alle regole di sottotipo, il tipo *Object* renderebbe valida l'invocazione incriminata
- l'errore in esecuzione è dovuto al fatto che gli array conservano il tipo con il quale sono stati creati (in questo caso, *Employee*)
- ogni scrittura nell'array controlla che l'oggetto assegnato sia di un tipo compatibile con quello dell'array

Per non incorrere in tali sorprese conviene **specificare il tipo desiderato** in tutte le invocazioni di metodi parametrici.

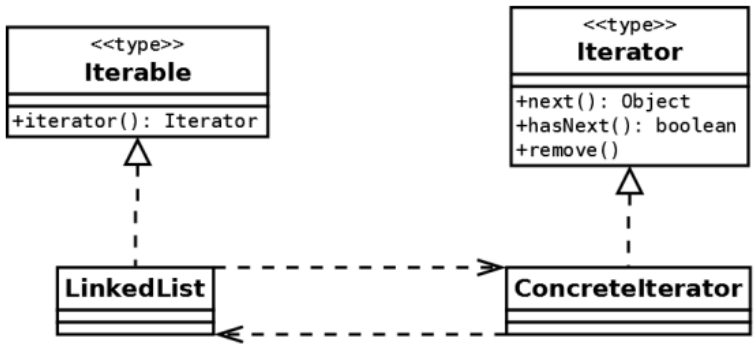
Iteratori

Il design pattern **Iterator** riguarda il modo di passare in rassegna gli elementi di una data collezione o insieme di oggetti, si suppone, cioè, che un determinato oggetto (contenitore) ne contenga altri.
Il contenitore vuole permettere agli utenti (client) di passare in rassegna tutti gli oggetti contenuti, senza però esporre la sua struttura interna. La soluzione di questo pattern consiste essenzialmente nel creare un oggetto, chiamato iteratore, che rappresenta un indice all'interno del contenitore.

Contesto:

1) Un oggetto (aggregato) contiene altri oggetti (elementi)
2) I clienti devono poter accedere a tutti gli elementi, uno alla volta
3) L'aggregato non deve esporre la sua struttura interna
4) Più clienti devono poter accedere contemporaneamente

Soluzione (illustriamo il pattern Iterator implementato nella libreria standard Java)**:**



- Il metodo ***next*** restituisce il prossimo elemento del contenitore, e contemporaneamente fa avanzare l'indice di una posizione, se viene chiamato quando non ci sono più elementi da visitare lancia un'eccezione (*NoSuchElementException*)
- Il metodo ***hasNext*** restituisce vero se c'è almeno un altro elemento del contenitore che deve ancora essere visitato.
- Il metodo ***remove***, non previsto dal pattern, elimina dal contenitore l'ultimo elemento che è stato restituito da *next*. Può essere chiamato **una sola volta dopo ogni chiamata a *next***. Questa operazione è facoltativa, ovvero, gli iteratori concreti sono liberi di non supportarla (in questo caso deve lanciare l'eccezione unchecked *UnsupportedOperationException*)
- L'interfaccia ***Iterable*** rappresenta il contenitore ed offre il metodo ***iterator*** che restituisce un nuovo oggetto iteratore.

LinkedList è una classe concreta che implementa le due interfacce e rappresenta una **lista doppiamente concatenata**, essa appartiene alla Java Collection Framework ed ha i seguenti metodi pubblici principali:

*boolean **add**(T x)*	Aggiunge x in coda alla lista e restituisce true
*boolean **contains**(Object x)*	Restituisce true se la lista contiene un oggetto y tale che x.equals(y) è vero
*boolean **remove**(Object x)*	Rimuove il primo oggetto della lista uguale a x Restituisce vero se ha trovato e rimosso l'oggetto
*int **size**()*	Restituisce la dimensione della lista
*void **addFirst**(T x)*	Aggiunge x in testa alla lista
*void **addLast**(T x)*	Equivalente ad *add(x)*, ma senza valore restituito
*T **removeFirst**()*	Rimuove e restituisce la testa della lista
*T **removeLast**()*	Rimuove e restituisce la coda della lista

8. Design by Contract

Programmazione tramite contratti

L'idea della programmazione guidata da contratti (proposta da Bertrand Meyer, creatore del linguaggio Eiffel) consiste nell'applicare al software, in particolare a quello orientato agli oggetti, la nozione comune di contratto: un accordo in cui le parti si assumono degli obblighi in cambio di benefici.

Il contratto può essere applicato anche a classi e interfacce, ma lo vedremo solo per i metodi. Applicato ad un **metodo** di una classe, un contratto specifica quale **compito** (o **postcondizione**) il metodo permette di svolgere e quali sono le **precondizioni** richieste. Dal punto di vista del client il compito svolto dal metodo è un beneficio mentre le precondizioni sono obblighi, differentemente, dal punto di vista del metodo il compito da svolgere è un obbligo e le precondizioni sono un beneficio (agevolano o consentono il compito).

Contratto di un metodo:

- La **precondizione** riguarda i valori passati al metodo come argomenti e/o lo stato dell'oggetto su cui viene invocato il metodo
- La **postcondizione** riguarda il valore restituito, il nuovo stato dell'oggetto e qualsiasi altro effetto collaterale.
- Viene considerato un effetto collaterale (side-effects) tutto quello che fa un metodo e che è visibile all'esterno, eccetto restituire un valore (esempi sono stampare un messaggio a video, scrivere in un file, terminare il programma, etc…)

Inoltre, il contratto può specificare come reagisce il metodo nel caso in cui le precondizioni non siano soddisfatte dal chiamante, analogamente ad una **penale** in un comune contratto.
In Java, la penale tipica consiste nel lancio di un'eccezione non verificata).

La **precondizione** deve essere **verificabile** da pare del client, ad esempio, un metodo che si interfaccia con una stampante non può prevedere come precondizione che la stampante abbia carta a sufficienza, perché questa non è una condizione che il client può verificare.

Violazione del contratto:

- La **precondizione** descrive l'uso corretto del metodo ed una sua violazione costituisce **errore di programmazione da parte del client**
- La **postcondizione** descrive l'effetto atteso del metodo ed una sua violazione indica un **errore di programmazione nel metodo stesso**

Documentare un contratto

In qualsiasi linguaggio di programmazione, solo una parte del contratto può essere espressa tramite codice, mentre il resto sarà indicato in linguaggio naturale con i commenti.

In un programma, ad esempio, il numero e tipo dei parametri formali di un metodo fanno parte delle precondizioni mentre il tipo di ritorno fa parte della postcondizione.

Un altro esempio è dato dalla dichiarazione "*throws …*". A prima vista, potrebbe sembrare che indichi il tipo di eccezione che viene sollevata in caso di violazione della precondizione, ma in realtà, la dichiarazione *throws* andrebbe usata solo per le eccezioni **verificate**, mentre la violazione di una precondizione richiede un'eccezione non verificata perché si tratta di un errore di programmazione. Dunque, la clausola *throws* usata correttamente, esprime parte della postcondizione; infatti, è responsabile della postcondizione descrivere le eventuali condizioni anomale che possono dare luogo ad eccezioni verificate.

In un contratto **ben definito**, la precondizione contiene tutte le proprietà che servono al metodo di svolgere il suo compito (cioè, per realizzare la sua postcondizione). Ad esempio, si considerino i seguenti contratti:

1) Il metodo *max* accetta due oggetti e restituisce il maggiore tra i due
   - Il contratto non è ben definito perché non è chiaro quale relazione d'ordine utilizzare
   - Una versione corretta: il metodo *max* accetta due oggetti di una classe dotata di ordinamento naturale e restituisce il maggiore tra i due
2) Il metodo *reverse* accetta una collezione e inverte l'ordine dei suoi elementi
   - Il contratto non è ben definito perché non tutte le collezioni sono ordinate; ad esempio, l'operazione non ha senso su un HashSet
   - Una versione corretta: il metodo *reverse* accetta una *lista* e inverte l'ordine dei suoi elementi

Contratto di Iterator

Presentiamo il contratto dei metodi dell'interfaccia *Iterator<T>*. Per un'interfaccia il contratto è particolarmente importante, perché rappresenta praticamente la sua vera ragione d'essere:

boolean hasNext()

- Precondizione:
  - Nessuna, tutte le invocazioni sono lecite
- Postcondizione:
  - Restituisce true se ci sono ancora elementi su cui iterare (cioè, se è lecito invocare *next*) e false altrimenti
  - Non modifica lo stato dell'iteratore

T next()

- Precondizione:
  - Ci sono ancora elementi su cui iterare (cioè, un'invocazione a *hasNext* restituirebbe true)
- Postcondizione:
  - Restituisce il prossimo oggetto della collezione
  - Fa avanzare l'iteratore all'oggetto successivo, se esiste
- Trattamento degli errori (penale)
  - Solleva *NoSuchElementException* (non verificata) se la precondizione è violata

void remove()

- Precondizione:
  - Prima di questa invocazione a *remove*, è stato invocato *next*, rispettando la sua precondizione
  - Dall'ultima invocazione a *next*, non è stato già chiamato *remove*
- Postcondizione:
  - Rimuove dalla collezione l'oggetto restituito dall'ultima chiamata a *next*
  - Non modifica lo stato dell'iteratore (cioè, non ha influenza su quale sarà il prossimo oggetto ad essere restituito da *next*)
- Trattamento degli errori (penale)
  - Solleva *IllegalStateException* (non verificata) se la precondizione è violata
- Inoltre, il metodo remove viene indicato come "opzionale" dalla documentazione
  - Ciò significa che le implementazioni di Iterator non sono obbligate a supportare questa funzionalità
  - Se un'implementazione non vuole supportarla, deve far lanciare al metodo *remove* l'eccezione *UnsupportedOperationException* (non verificata)

Come esempio riportiamo un esercizio d'esame (21/04/2008):

4) Individuare l'output del seguente programma. Dire se la classe *CrazyIterator* rispetta il contratto dell'interfaccia *Iterator*. In caso negativo, giustificare la risposta.

```java
public class CrazyIterator implements Iterator {
    private int n = 0;
    public Object next() { // nessuna violazione
        int j;
        while (true) {
            for (j=2; j<=n/2 ;j++)
                if (n % j == 0) break;
            if (j > n/2) break;
            n++;
        }
        return new Integer(n);
    }
    public boolean hasNext() { // viola il contratto
        n++; // modifica l'iteratore, cosa che hasNext
             // non ha diritto di fare
        return true;
    }
    public void remove() { // viola il contratto
        throw new RuntimeException(); // l'eccezione da lanciare
        // nel caso in cui non si vuole implementare remove deve
        //essere UnsupportedOperationException
    }

    public static void main(String[] args) {
        Iterator i = new CrazyIterator();
        while (i.hasNext() && (Integer)i.next()<10) {
            System.out.println(i.next());
        }
    } // Output: 1 2 3 5 7
}
```

Contratti ed overriding

Il contratto di un metodo andrebbe considerato vincolante anche per le eventuali ridefinizioni del metodo (overriding).

Il contratto di un metodo ridefinito in una sottoclasse deve assicurare il **principio di sostituibilità**:

- Le chiamate fatte al metodo originario rispettando il suo contratto devono continuare ad essere corrette anche rispetto al contratto ridefinito in una sottoclasse

Questo è un caso particolare del noto **principio di sostituzione di Liskov**:

- Se un client usa una classe *A*, deve poter usare allo stesso modo le sottoclassi di *A*, senza neppure conoscerle.

Quindi, ogni ridefinizione del contratto dovrebbe offrire al client almeno gli stessi benefici, richiedendo al più gli stessi obblighi. In altre parole, un overriding può rafforzare la postcondizione (garantire di più) e indebolire la precondizione (richiedere di meno). Tale condizione prende il nome di **regola contro-variante**, perché la precondizione può variare in modo opposto alla postcondizione.

Forniamo un esempio per spiegare le regole dell'overriding in Java (derivano in buona parte dal principio di sostituibilità):

- Dato un metodo con la seguente intestazione: `Vis T foo(U1, …, Un) throws E1, …, Em`
- Può cambiare in un overriding in: `Vis' T' foo(U1, …, Un) throws F1, …, Fk`
    - *Vis'* può essere più ampia di *Vis*
    - *T'* può essere sottotipo di *T*
    - Per ogni *i,j = 1, …, k, Fi* è sottotipo di qualche *Ej*

Le regole dell'overriding in Java rispecchiano solo una metà della regola contro-variante; infatti, il **tipo di ritorno** di un metodo può diventare **più specifico** nell'overriding, rafforzando quindi la postcondizione. Invece, il **tipo dei parametri non può cambiare**, mentre la regola contro-variante prevederebbe che potessero diventare più generali.

Ad esempio, consideriamo il seguente contratto per il metodo con intestazione

```
public double avg(String str, char ch)
```

- Precondizione: la stringa *str* non è null e non è vuota; il carattere *ch* è presente in *str*
- Postcondizione: restituisce il numero di occorrenze di *ch* in *str*, diviso la lunghezza di *str*

I seguenti contratti sono overriding validi:

1) Precondizione: la stringa *str* non è null
   Postcondizione: se la stringa è vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di *ch* in *str*, diviso la lunghezza di *str*
2) Precondizione: nessuna
   Postcondizione: se la stringa è null oppure vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di *ch* in *str*, diviso la lunghezza di *str*

Mentre non lo è:

3) Precondizione: la stringa *str* non è null e non è vuota
   Postcondizione: restituisce il numero di occorrenze di *ch* in *str*

Contratti in Javadoc

Javadoc è un tool che estrae documentazione dai sorgenti Java e la rende disponibile in vari formati, tra cui l'HTML. Le informazioni le estrae dalle dichiarazioni e da alcuni commenti speciali, racchiusi tra */*** e **/* dove all'interno, si possono utilizzare marcatori (tag) per strutturare la documentazione.

In particolare, il contratto di un metodo andrebbe indicato in un commento che precede il metodo, utilizzando almeno i tag *@param*, *@return* e *@throws*, ad esempio, consideriamo un metodo che calcola la **radice quadrata** di un numero dato, dove:

- La **precondizione** consiste nel fatto che l'argomento deve essere un numero non negativo
- La **postcondizione** assicura che il valore restituito è la radice quadrata dell'argomento
- La reazione all'errore (**penale**) consiste nel lancio dell'eccezione *IllegalArgumentException*

Utilizzando Javadoc, il suo contratto può essere sinteticamente indicato come segue:

```
/**
 * Calcola la radice quadrata.
 *
 * @param x numero non-negativo di cui si vuole calcolare la radice
 * @return la radice quadrata di x
 * @throws IllegalArgumentException se x è negativo
 */
public double sqrt(double x) {
    if (x<0) throw new IllegalArgumentException();
    ...
}
```

In alcuni casi, è conveniente distinguere due parti del contratto:

- La parte **generale**, che si applica anche a tutte le possibili ridefinizioni del metodo
- La parte **locale**, che si applica solo alla versione originale del metodo, ma non costituisce un obbligo per le ridefinizioni

Ne è un buon esempio il **contratto di equals** (lo presenteremo parafrasando la documentazione ufficiale di Java) poiché il metodo equals è destinato ad essere ridefinito nelle sottoclassi, il contratto è diviso in parte generale e parte locale, quest'ultima descrive il comportamento della versione originale presente in Object, ma non è vincolante per le sottoclassi di Object:

Parte generale

- Precondizione: nessuna, tutte le invocazioni sono lecite
- Postcondizione:
  - Il metodo rappresenta una relazione di equivalenza tra istanze non nulle
  - L'invocazione *x.equals(y)* restituisce vero se y è diverso da null ed è considerato "equivalente" a x
  - Invocazioni ripetute di equals su oggetti il cui stato non è cambiato devono avere lo stesso risultato (coerenza temporale)

Parte locale

- Precondizione: nessuna
- Postcondizione: l'invocazione *x.equals(y)* è equivalente a *x==y* (quando x non sia null)

9. Confronto e ordinamento tra oggetti

L'interfaccia Comparable

La libreria standard Java fornisce due interfacce per impostare un criterio di ordinamento. La prima che vedremo è l'interfaccia *Comparable*:

```
public interface Comparable<T> {
    public int compareTo(T x);
}
```

Semplicemente, quando si vuole dotare una classe di un criterio di ordinamento, si può far implementare alla classe l'interfaccia *Comparable*. L'uso di questa interfaccia è indicato quando la classe possiede **un unico criterio** di ordinamento naturale.

Il contratto del metodo *compareTo* con argomento *x* è il seguente:

- Precondizione: l'oggetto *x* è confrontabile con *this*
- Postcondizione:

  restituisce:
  - Un valore negativo se *this* è minore di *x*
  - 0 se *this* è uguale a *x*
  - Un valore positivo se *this* è maggiore di *x*
- Il metodo dovrebbe lanciare l'eccezione (non verificata) *ClassCastException* se riceve un oggetto che, a causa del suo tipo effettivo, non è confrontabile con *this*

Da esempio, consideriamo la classe *Employee*, con campi nome (*String*) e salario (int). Se siamo certi che gli *Employee* saranno sempre confrontati alfabeticamente per nome, faremo in modo che *Employee* implementi l'interfaccia *Comparable*, come segue:

```
public class Employee implements Comparable<Employee> {
    private int salary;
    private String name;

    @Override
    public int compareTo(Employee x) {
        return name.compareTo(x.name);
    }// sfruttiamo il fatto che String implementi
    // a sua volta Comparable<String>
}
```

L'interfaccia Comparator

In alternativa, si può realizzare una seconda classe che implementi l'interfaccia *Comparator*:

```
public interface Comparator<T> {
    public int compare(T x, T y);
}
```

Il contratto del metodo *compare* di *Comparator* è analogo a quello di *compareTo* di *Comparable*:

- Precondizione: l'oggetto *x* è confrontabile con *y*
- Postcondizione:
  restituisce:
  - Un valore negativo se *x* è minore di *y*
  - 0 se *x* è uguale a *y*
  - Un valore positivo se *x* è maggiore di *y*

L'uso di *Comparator* è indicato quando la classe da ordinare **non ha un unico criterio** di ordinamento naturale, oppure la classe da ordinare è già stata realizzata e **non si può o non si vuole modificarla**.

Supponiamo, ad esempio di voler offrire due diversi criteri di ordinamento per gli *Employee*: per nome e per salario. Per farlo, dobbiamo utilizzare l'interfaccia *Comparator*, come segue:

```
public class Employee {
    private int salary;
    private String name;
    public static final Comparator<Employee> comparatorByName = new Comparator<>() {
        public int compare(Employee a, Employee b) {
            return a.name.compareTo(b.name);
        }
    };
    public static final Comparator<Employee> comparatorBySalary = new Comparator<>() {
        ...
        // in alternativa alla classe anonima si potrebbe usare una lambda-espressione
    };
}
```

Si noti che essendo tutti gli oggetti della classe anonima equivalenti tra di loro non ha senso avere un istanza per ogni *Comparator*, essendo infatti questa classe stateless (senza attributi) per questo utilizziamo il modificatore *static* e non seguiamo la stessa soluzione di *Comparable*.

Il caso della classe String

Le stringhe sono dotate di un ordinamento naturale che è quello alfabetico (o lessicografico). La classe String fornisce questo criterio di confronto implementando *Comparable*.

Nell'ordinamento naturale delle stringhe, le lettere maiuscole vengono prima delle minuscole (si basa sui codici UNICODE dei caratteri), ma può essere utile anche ordinare stringhe senza considerare la distinzione tra minuscole e maiuscole (case insensitive).

Poiché non è possibile implementare *Comparable* in due modi diversi, questo criterio di confronto alternativo deve essere fornito da un oggetto di tipo Comparator; difatti, nella classe String troviamo la seguente costante:

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER;
```

Cheat: se un esercizio dice di avere un criterio di ordinamento naturale allora deve estendere *Comparable*. Mentre se dice di avere un altro tipo di ordinamento allora si usa *Comparator* nel modo già visto.

Proprietà dell'ordinamento

Affinché l'implementazione di *Comparable* o *Comparator* definisca effettivamente un criterio di ordinamento tra oggetti, essa **dovrà** rispettare le seguenti proprietà:

- Dato un numero reale *a*, definiamo la funzione segno *sgn(a)*: $sgn(a) = \begin{cases} 1 \ \ se \ \ a > 0 \\ 0 \ \ se \ \ a = 0 \\ -1 \ se \ \ a < 0 \end{cases}$
- Dati tre oggetti *x, y* e *z*, appartenenti ad una classe che implementa *Comparable* (condizioni analoghe varranno per le implementazioni di *Comparator*), devono valere le seguenti condizioni (si noti la somiglianza con le proprietà di una relazione d'ordine):
    1) $sgn\big(x.compareTo(y)\big) == -sgn\big(y.compareTo(x)\big)$ (**riflessività e antisimmetria**)
    2) Se $x.compareTo(y) < 0$ e $y.compareTo(z) < 0$ allora $x.compareTo(z) < 0$ (**transitività**)
    3) Se $x.compareTo(y) == 0$ allora $sgn\big(x.compareTo(z)\big) == sgn\big(y.compareTo(z)\big)$

È preferibile, ma non obbligatorio, che le implementazioni di *Comparable* e *Comparator* siano coerenti con *equals*. Nel caso di Comparable, questo vuol dire che $x.compareTo(y) == 0 \Leftrightarrow x.equals(y) == true$ (ragionamento analogo per *Comparator*).

Nota: il comparatore String.CASE_INSENSITIVE_ORDER non è coerente con equals (per rimarcare il fatto che non è obbligatoria questa coerenza con equals)

Uso di comparatori per ordinare array e liste

Nell'API Java sono presenti metodi che utilizzano le interfacce *Comparable* e *Comparator* per fornire algoritmi di ordinamento di array e liste.

Per quanto riguarda gli array, tali metodi si trovano nella classe *java.util.**Arrays***, una classe che contiene solo metodi statici, in particolare:

1) `public static void sort(Object[] a)`
   Ordina l'array *a* in senso non-decrescente, in base all'ordinamento naturale tra i suoi elementi, ovvero, suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia *Comparable*.
2) `public static <T> void sort(T[] a, Comparator<T> c)`
   Ordina l'array *a* in senso non-decrescente, in base all'ordinamento indotto dal comparatore *c*

In entrambi i casi, l'ordinamento è **inplace** e **stabile**, cioè, l'array viene modificato senza utilizzare strutture di appoggio e gli elementi equivalenti secondo l'ordinamento mantengono l'ordine che avevano originariamente (l'algoritmo usato è una versione ottimizzata del merge sort)

Per quanto riguarda le liste, i metodi di ordinamento si trovano nella classe *java.util.**Collections***, una classe che contiene solo metodi statici, di nostro interesse sono i seguenti due:

1) `public static void sort(List l)`
   ordina la lista *l* in senso non-decrescente, in base all'ordinamento naturale tra i suoi elementi, ovvero, suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia Comparable.
2) `public static void sort(List l, Comparator c)`
   ordina la lista *l* in senso non-decrescente, in base all'ordinamento indotto dal comparatore *c*

Come per gli array, l'ordinamento è inplace e stabile. L'algoritmo usato è una versione ottimizzata del merge sort. Nota: per semplicità sono state presentate le versioni grezze dei metodi

10. Parametri di tipo con limiti

Tipi parametrici e relazioni di sottotipo

Supponiamo di voler scrivere un metodo statico che accetta una lista di *Employee* e restituisce la somma dei loro salari:

```
public static int getTotalSalary(List<Employee> l) {
    int tot = 0;
    for (Employee e: l)
        tot += e.getSalary();
    return tot;
}
```

Questa versione non parametrica non accetta però sottoclassi di Employee poiché *List<Manager>* non è sottotipo di *List<Employee>*. Infatti, anche se una delle regole di sottotipo afferma che se *A* è sottotipo di *B*, array di *A* è sottotipo di array di *B*, ma questo principio non vale per i tipi parametrici.

Ovvero, se *X* è una classe con un parametro di tipo, ed *A* e *B* sono due tipi tali che *A* è sottotipo di *B*, non è vero che *X<A>* è sottotipo di *X<B>*.

Confrontiamo il comportamento di array e collezioni, rispetto ai sottotipi considerando il seguente frammento di codice:

```
Manager[] managers = new Manager[10];
Employee[] employees = managers;
employees[0] = new Employee(...);
```

- Questo frammento **compila correttamente**, grazie alle regole di sottotipo per array
- Tuttavia, il terzo rigo provoca un'**eccezione a runtime**; infatti, non è possibile aggiungere un *Employee* ad un array che è stato dichiarato come *Manager[]*

Sviluppiamo un esempio simile al precedente, sostituendo gli array con oggetti di tipo ArrayList:

```
ArrayList<Manager> managers = new ArrayList<Manager>();
ArrayList<Employee> employees = managers;
employees.add(new Employee(...));
```

- Questa volta, il frammento di codice provoca un **errore di compilazione** (che è preferibile all'errore in runtime poiché più facile da individuare e quindi correggere.
- In particolare, l'errore si verifica al secondo rigo, perché *ArrayList<Manager>* **non** è sottotipo di *ArrayList<Employee>* (e quindi non assegnabile)

Concludiamo che il sistema dei tipi parametrici è **più robusto** di quello degli array, relativamente agli errori di tipo; infatti, le regole dei tipi parametrici garantiscono che, se il programma viene compilato senza warning e se non utilizza array o cast, non possono verificarsi errori di tipo al runtime.

Limiti superiori

Ritornando all'esempio di *getTotalSalary*, per risolvere il problema di non poter passargli una lista di *Manager*, possiamo utilizzare un **parametro di tipo con limite superiore**.

Quando si dichiara un parametro di tipo, appartenete ad una classe oppure ad un metodo, si può specificare che quel parametro potrà assumere come valore **solo sottotipi di un dato tipo**. Questo tipo viene chiamato limite superiore per il parametro in questione.

Un parametro di tipo può anche avere più limiti superiori simultaneamente, in questo caso, il parametro attuale di tipo dovrà essere sottotipo di ciascuno dei limiti superiori (questo significa anche che il primo tipo può essere una classe, mentre gli eventuali successivi limiti devono essere interfacce poiché non avrebbe senso che il parametro attuale estendesse simultaneamente due classi). La sintassi per indicare uno o più

limiti superiori è la seguente: *<T extends U1 & U2 & …>* (si ricorda che la virgola viene usata per separare i parametri di tipo, per cui è stato deciso di usare la & come separatore di limiti).

Quindi, otteniamo la seguente versione di *getTotalSalary*:

```
public static <T extends Employee> int getTotSalary(List<T> l) {
    int tot = 0;
    for (T e: l)
        tot += e.getSalary();
    return tot;
}
```

- È possibile invocare questa versione passando una lista di qualsiasi sottotipo di *Employee*
- I riferimenti di tipo *T* (come *e* nell'esempio) sono trattati dal compilatore come se fossero di tipo *Employee*

La versione parametrica di Comparable e Comparator

Riprendiamo le due interfacce *Comparable* e *Comparator*:

```
public interface Comparable<T> {
    public int compareTo(T x);
}
```

```
public interface Comparator<T> {
    public int compare(T x, T y);
}
```

Come si vede, il parametro di tipo permette di specificare che tipo di oggetti tali metodi sono in grado di confrontare; ad esempio, se la classe *Employee* intende fornire un ordinamento naturale tra impiegati essa si presenterà come segue: `public class Employee implements Comparable<Employee>`

Supponiamo di voler scrivere un metodo statico che accetta una lista di oggetti dotati di ordinamento naturale, e restituisce l'elemento minimo della lista.
Proviamo la seguente versione non parametrica:

```
public static Object getMin(List<Comparable<Object>> l) {
    Object min = null;
    for (Comparable<Object> x: l)
        if (min == null || x.compareTo(min) < 0)
    min = x;
    return min;
}
```

- Il problema con questa versione è che possiamo passare a *getMin* **solo** una *List<Comparable<Object>>*
- Anche se *Employee* implementasse *Comparable<Object>* (che già sarebbe strano), comunque *List<Employee>* non sarebbe sottotipo di *List<Comparable<Object>>*

Creiamo una nuova versione di *getMin*: chiamiamo *T* il tipo degli elementi della lisa e stabiliamo che *T* debba essere sottotipo di *Comparable*; inoltre, siccome Comparable è a sua volta parametrica, bisogna specificare anche il suo parametro di tipo.
Otteniamo quindi la seguente soluzione:

```
public static <T extends Comparable<T>> T getMin(List<T> l) {
    T min = null;
    for (T x: l)
        if (min == null || x.compareTo(min) < 0)
    min = x;
    return min;
}
```

- Questa versione ha anche il vantaggio che il tipo restituito è lo stesso di quello della lista
- Se *Employee* implementa *Comparable<Employee>*, è possibile passare al metodo precedente una lista di Employee

Quest'ultima versione ha comunque un "problema", infatti, supponendo che la classe *Manager* estenda *Employee*, e non implementi direttamente nessuna versione di *Comparable* non è possibile passare al metodo *getMin* una lista di *Manager* perché la classe *Manager* non implementa *Comparable<Manager>* (neanche indirettamente), anche se *Manager* implementa indirettamente *Comparable<Employee>*.
Questa limitazione si può ovviare con il parametro jolly.

Il parametro di tipo jolly

Le regole appena descritte per la relazione di sottotipo tra tipi parametrici comportano, tra l'altro, che *List<Object>* **non** sia il supertipo comune a tutte le liste; quindi, sembrerebbe che, se un metodo intende accettare liste di ogni tipo, esso debba necessariamente essere parametrico.

Invece, **il parametro di tipo jolly**, rappresentato sintatticamente da un punto interrogativo, permette di raggiungere lo stesso scopo senza utilizzare parametri di tipo. Ovvero: ***List<?>*** è il supertipo comune a tutte le versioni di *List*.

Il parametro di tipo jolly si può usare solamente come **parametro attuale di tipo**, ed intuitivamente rappresenta un tipo sconosciuto. Non c'è nessuna relazione tra due diverse occorrenze del parametro jolly; ad esempio, il seguente metodo accetta **due liste dello stesso tipo**

```
public static <T> void foo(List<T> l1, List<T> l2) { ... }
```

mentre il seguente metodo accetta **due liste qualsiasi**

```
public static void bar(List<?> l1, List<?> l2) { ... }
```

Supponiamo che *A<T>* sia una classe parametrica. Dichiarando un riferimento di tipo *A<?>*, è come se le occorrenze di *T* all'interno di *A* diventassero "?", cioè, "tipo sconosciuto". Questo pone delle limitazioni sull'uso che si può fare di quel riferimento.

In particolare, se un metodo della classe *A<T>* **accetta un argomento di tipo *T***, usando un riferimento di tipo *A<?>* potremo passare a quel metodo solamente **null**; infatti, non sapendo qual è il tipo concreto che quel metodo si aspetta, null è l'unico valore che è lecito passargli.
Se invece un metodo della classe *A<T>* **restituisce un valore di tipo *T*,** chiamandolo con un riferimento di tipo *A<?>* potremo assegnare il valore restituito solamente ad un riferimento di tipo **Object**; infatti Object è l'unico tipo che sicuramente accetta qualunque tipo sconosciuto (non primitivo).

Prendiamo, ad esempio, il seguente metodo che estrae la testa di una LinkedList e la inserisce in coda:

```
public static <T> void moveHeadToTail(LinkedList<T> l) {
    T head = l.removeFirst();
    l.addLast(head);
}
```

Se sostituiamo il parametro *T* con *?*, **non** possiamo svolgere lo stesso compito:

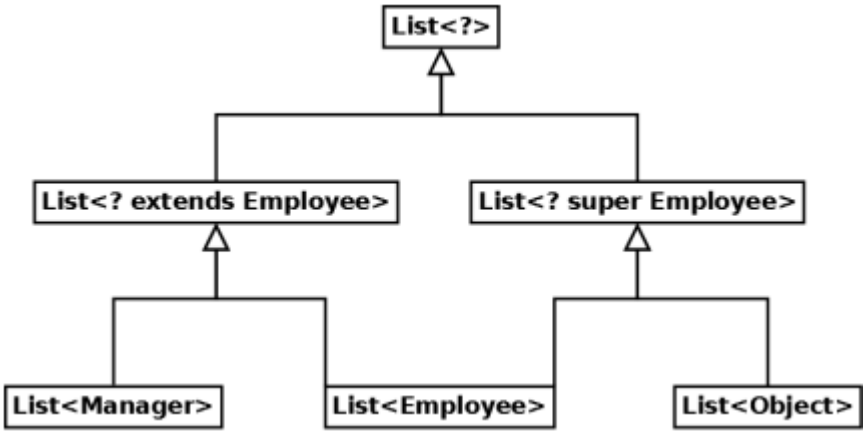
```
public static void moveHeadToTail(LinkedList<?> l) {
    Object head = l.removeFirst(); // si può assegnare ad Object
    l.addLast(head); // Errore di compilazione: accetta solo null
}
```

Jolly con limiti superiori e inferiori

Come i normali parametri di tipo, anche il parametro di tipo jolly può avere un limite superiore (ma non più di uno). Ad esempio, *List<? extends Employee>* è una lista di tipo sconosciuto che estende *Employee*, più precisamente è il **supertipo comune** a tutte le liste il cui parametro di tipo estende *Employee*.

A differenza dei normali parametri di tipo, il tipo jolly può anche avere un **limite inferiore**, ad esempio, *List<? super Employee>* rappresenta un lista di un tipo sconosciuto che è **supertipo** di *Employee* (come *Person* o *Object*).

La seguente figura rappresenta la relazione di sottotipo tra diverse versione di *List*:



- Diamo un esempio per comprendere meglio le relazioni di sottotipo. Siano:
  ```
  1   List<? extends Employee>
  2   List<? extends Manager>
  ```
  La 2 è sempre assegnabile alla 1 essendo la 2 rappresentante un insieme di valori che è sottotipo dei valori rappresentati dalla 1 (i tipi che estendono *Manager* sono sicuramente assegnabili ai tipi che estendono *Employee*)

Consideriamo nuovamente il metodo *getMin*. Il parametro di tipo jolly permette di dichiarare il metodo in modo tale da accettare una lista di *Manager*, anche se *Manager* implementa *Comparable<Employee>*:

```
public static <T extends Comparable<? super T>> T getMin(List<T> l) { ... }
```

Riassumendo, consideriamo le seguenti intestazioni per *getMin*:

```
Object getMin(List<Comparable<Object>> l)              // non accetta List<Employee>
<T> T getMin(List<Comparable<T>> l)                    // non accetta List<Employee>
<T extends Comparable<T>> T getMin(List<T> l)          // non accetta List<Manager>
<T extends Comparable<? super T>> T getMin(List<T> l)  // ok
```

Nota: Quest'ulteriore proposta, che usa un secondo parametro di tipo invece di un jolly, non compila a causa di una limitazione del sistema dei tipi:

```
<T, S extends T & Comparable<T>> S getMin(List<S> l)
// Messaggio di errore: a type variable may not be followed by other bounds
```

Esempio 1: Realizziamo un metodo che accetta una collezione di valori numerici e ne restituisce la somma. Sappiamo che le collezioni non possono contenere tipi primitivi, quindi ci riferiamo alle classi wrapper di tipo numerico, le quali estendono *Number* che offre il metodo *doubleValue*, il quale converte in double questo valore numerico qualunque sia il suo tipo.

```
public static double getSum(Collection<? extends Number> c) {
    double sum = 0.0;
    for (Number n: c)
        sum += n.doubleValue();
    return sum;
}
```

- Il parametro jolly ci ha consentito di scrivere un metodo *getSum* che non è parametrico
- Il metodo *getSum* accetterà una collezione di qualunque tipo wrapper numerico.

Esempio2: Realizziamo un metodo che accetta una collezione di oggetti confrontabili e una coppia di oggetti (ipotetica classe *Pair<T>*) e modifica la coppia in modo che contenga l'oggetto minimo e quello

massimo della collezione:

```
public static <T extends Comparable<? super T>>
    void getMinMax(Collection<T/*1)*/> c, Pair<? super T/*2)*/> p)
{
    T min = null, max = null;
    for (T x: c) {
        if (min==null || x.compareTo(min)<0)
            min = x;
        if (max==null || x.compareTo(max)>0)
            max = x;
    }
    p.setFirst(min);
    p.setSecond(max);
}
```

1) Mettere *<T extends Comparable>* qui sarebbe un errore poiché accetterebbe solo tipi che implementano l'interfaccia *Comparable*.
2) Mettere solo *<T>* andrebbe anche bene ma sarebbe più limitato; infatti, nulla vieta al cliente di voler assegnare delle Stringhe, ad esempio, in una *Pair* di *Object*.

Come abbiamo visto per il tipo jolly senza limiti, l'uso del parametro jolly con limiti superiori o inferiori impone determinate condizioni sulle chiamate ai metodi. Supponiamo che *A<T>* sia una classe parametrica, la seguente tabella riassume le limitazioni che valgono per un riferimento di tipo *A<?>*, *A<? extends B>*, oppure *A<? super B>*, rispetto ad un metodo di *A* che accetta un argomento di tipo *T*, oppure restituisce un valore di tipo *T*:

tipo del riferimento	**cosa si può passare a *f(T)***	**a cosa si può assegnare *T f()***
A<?>	solo null	solo ad Object
A<? extends B>	solo null	a *B* e ai suoi supertipi
A<? super B>	*B* e suoi sottotipi	Solo ad Object

11. Erasure

Implementare i Generics

Ogni linguaggio che supporta i parametri di tipo (polimorfismo parametrico) ha un suo modo di implementare i generics. Prendiamo ad esempio C++ che ha un meccanismo simile alla programmazione generica di java ma più "naturale", chiamato **template**:

```
template<class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair(const T1& a, const T2& b) : first(a), second(b) { }
    ...
};
```

- Quando il compilatore C++ trova un riferimento ad una versione concreta di un template, come *pair<string, employee>*, esso **istanzia** una nuova copia della classe *pair*, con *string* al posto di *T1* ed employee al posto di *T2* (pesa molto sul file oggetto poiché questo meccanismo crea praticamente una copia dello stesso codice che cambia solo per il tipo effettivo).
- Questo approccio, detto **reificazione** (concretizzazione) **a tempo di compilazione**, è diametralmente opposto a quello di Java che sfrutta, invece, il meccanismo della **cancellazione** (erasure) dove il compilatore usa i generics per fare un type checking più accurato, poi scarta l'informazione.

La cancellazione

Approfondiamo il meccanismo implementativo scelto da Java e le limitazione che questo meccanismo comporta sull'uso dei parametri di tipo. Il meccanismo che supporta la programmazione generica prende il nome di **cancellazione** (in inglese **erasure**).

Il principio di funzionamento dell'erasure è il seguente:

1) I parametri di tipo vengono usati dal compilatore per effettuare i dovuti controlli (type checking)
2) Poi, tutti i parametri di tipo vengono rimossi (cancellati, appunto) e sostituiti da Object, oppure dal primo limite superiore del parametro in questione, se presente.
3) Il parametro di tipo jolly viene semplicemente rimosso
4) Poiché si sono persi i parametri di tipo, in conseguenza della cancellazione, vengono inseriti degli opportuni cast, per ripristinare la coerenza tra i tipi

Di conseguenza, nel **bytecode** risultato della compilazione **non c'è più traccia dei parametri di tipo**; ovvero, in fase di esecuzione i tipi parametrici sono scomparsi.

Questo metodo dell'erasure comporta le seguenti limitazioni nell'uso dei generics:

- **Non è possibile utilizzare un parametro di tipo per istanziare oggetti**
  - *new T()* (errore di compilazione): infatti, al tempo di esecuzione tale istruzione diventerebbe *new Object()*, che sicuramente non è quello che il programmatore intendeva ottenere; d'altronde, è possibile utilizzare un parametro di tipo per istanziare una classe concreta, come in *new LinkedList<T>()*
  - Il parametro di tipo jolly, invece, non può essere utilizzato neanche per istanziare una classe concreta.
  - *new LinkedList<?>()* (errore di compilazione): quest'ultima istruzione corrisponderebbe alla richiesta di creare una lista di oggetti di tipo sconosciuto. Ricordiamo che il parametro di tipo jolly serve per avere riferimenti in grado di puntare a diverse versioni di una classe parametrica.
- **Non è possibile istanziare un array di tipo parametrico**
  - *new T[10]* (errore di compilazione): gli array ricordano il tipo con il quale sono stati creati; quindi, questo array sarebbe a tutti gli effetti di tipo *Object*; una possibile soluzione consiste nell'istanziare una lista di tipo *T*, invece di un array
  - È, invece, possibile dichiarare un riferimento di tipo array di tipo parametrico: *T[] a;* Questo costrutto è utile, ad esempio, come parametro formale di un metodo, per accettare array di qualsiasi tipo ed associare il tipo dell'array a quello di altri parametri, oppure al tipo di ritorno

I parametri di tipo presentano anche una serie di problemi legati all'overloading:

- **Non è possibile usare un parametro di tipo per distinguere due versioni di un metodo**
  - Consideriamo la classe *Pair<S,T>*, che rappresenta una coppia di elementi di due tipi diversi, si potrebbe essere tentati di realizzarla secondo il seguente schema:

```
public class Pair<S, T> {
    private S first;
    private T second;
    public void setValue(S x) { first = x; }
    public void setValue(T x) { second = x; }
    ...              // ^ errore di compilazione
}
```

  - Questa soluzione non funziona, perché entrambi i parametri di tipo, dopo il type checking, diventeranno Object, e quindi l'overloading diventerà non valido.

- **Dopo l'erasure, un metodo parametrico potrebbe andare in conflitto con uno non parametrico**
  - Ad esempio, non possiamo avere i seguenti metodi nella sessa classe:
    ```
    public <T> void f(T t) { ... }
    public void f(Object o) { ... } // errore di compilazione
    ```
  - Invece, i seguenti metodi possono coesistere:
    ```
    public <T> void f(T t) { ... }
    public void f(String s) { ... }
    ```
- **Non è possibile utilizzare un parametro di tipo per selezionare una determinata versione di un metodo in overloading**
  - Ad esempio, consideriamo i seguenti metodi:
    ```
    public void f(String s) { ... }
    public void f(Object o) { ... }
    public <T> void g(T x) { f(x); }
    ```
  - Indipendentemente dal valore assunto dal parametro di tipo *T*, il metodo *g* chiamerà sempre *f(Object)* anche se *g* sarà chiamato con *T = String*
  - Infatti, la risoluzione dell'overloading avviene a tempo di compilazione, quando ancora non si conosce il valore che *T* assumerà (il compilatore non può che assumere *T = Object*
  - Una possibile soluzione potrebbe essere quella di scrivere la funzione *g* come segue:
    ```
    public <T> void g(T x) {
        if (x istanceof String)
            f((String) x);
        else
            f(x);
    }
    ```

L'erasure comporta delle limitazioni anche per le conversioni o l'uso dell'operatore istanceof

- **I parametri di tipo non vanno usati per effettuare conversioni esplicite (cast)**
  - In particolare, abbiamo un warning con: *(T) x;* poiché a runtime diventa un cast verso *Object* (o verso il primo limite superiore di *T*, se presente)
  - Qualunque cast verso un tipo parametrico (un parametro di tipo oppure una classe o interfaccia parametrica) produce un warning in compilazione: *(LinkedList<String>) x*
  - Se è necessario effettuare un cast, lo si può fare usando il parametro jolly: *(LinkedList<?> x*
  - Esempi di conversioni:
    ```
    (List<String>) new Object()
    // Warning, poi eccezione a run-time perché il tipo
    // effettivo (Object) non è sottotipo di List

    (List<String>) (List<?>) new LinkedList<Integer>()
    // Warning, nessuna eccezione a run-time;
    // con questa sequenza di conversioni possiamo inserire
    // una stringa in una lista di interi (cazzi nostri...)

    (List<String>) new LinkedList<Integer>()
    // Errore di compilazione perché LinkedList<Integer>
    // non è sottotipo di List<String>
    ```
- **Non si può applicare instanceof a un parametro di tipo o a una classe parametrica**
  - Esempio, *x instanceof T* è errore in compilazione; poiché a runtime diventerebbe *x instanceof Object*; quindi, il risultato sarebbe sempre true (tranne che per null)
  - Anche *x instanceof List<String>* porta ad un errore di compilazione. Questo contesto è uno dei pochi casi in cui è opportuno usare la versione grezza *x instanceof List*

Confronto cancellazione-reificazione

Vantaggi della **reificazione** (C++, C#):

- **Espressività**: si può fare con un parametro di tipo tutto quello che si può fare con un tipo concreto

Vantaggi della **cancellazione** (Java):

- **Evita il code bloating**: ripetizione, nell'eseguibile o in memoria, di codice simile
- **Supporta la compilazione separata**: cosa che non può fare il C++ poiché, per come implementati i template, questi vanno interamente in file .h; quindi, non si può compilare il template separatamente dai file sorgenti che lo utilizzano (devo ogni volta compilare anche il .h ogni volta che questo viene usato nei file sorgenti).

Concludiamo con un esempio di un metodo già esaminato della classe *java.util.Collections*:

```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> l)
```

ma che ora è possibile analizzare in dettaglio:

- Per individuare l'elemento minimo di una collezione, il metodo *min* ha bisogno che gli elementi della collezione siano mutuamente confrontabili tramite l'interfaccia *Comparable*; quindi, il parametro di tipo *T*, che rappresenta il tipo degli elementi della collezione, ha come limite superiore *Comparable<? Super T>*
- Inoltre, prima che esistessero i tipi parametrici in Java, questo metodo era già presente, con la firma: `public static Object min(Collection l)`. Ora è chiaro l'uso del limite superiore *<T extends Object &…>*, il quale fa sì che, dopo la cancellazione, la nuova firma coincida con la vecchia, a vantaggio della compatibilità con il codice preesistente
- Infine, il tipo del parametro *Collection<? extends T>* offre la garanzia che la collezione passata al metodo *min* non sarà modificata

12. Java Collection Framework

JCF e il ciclo enhanced-for

Il Java Collection Framework (JCF) è una parte della libreria standard dedicata alle collezioni, intese come classi deputate a contenere altri oggetti; questa libreria offre strutture dati di supporto, molto utili alla programmazione, come liste, array di dimensione dinamica, insiemi, mappe associative (anche chiamate dizionari) e code.

Le classi e interfacce del JCF si dividono in due gerarchie: quella che si diparte dall'interfaccia **Collection** e quella che si diparte da **Map**; inoltre, la classe Collection**s** (con la s finale) contiene numerosi algoritmi di supporto, ad esempio, metodi che effettuano l'ordinamento.

L'interfaccia *Collection* estende la versione parametrica di *Iterable*, della quale riportiamo la definizione parametrica qui di seguito:

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
    public E next();
    public boolean hasNext();
    public void remove();
}
```

Descriviamo ora il ciclo enhanced for o anche detto **for each**, con un esempio:

```
String[] array = {"uno", "due", "tre"};
for (String s: array)
    System.out.println(s);
```

Il ciclo precedente stampa tutte le stringhe contenute nell'array, una per rigo. Questa nuova forma di ciclo permette di iterare su un array senza dover esplicitamente utilizzare un indice e quindi senza il rischio di sbagliare gli estremi dell'iterazione.

Oltre che per gli array, il ciclo for-each funziona anche su tutti gli oggetti che implementano *Iterable<E>*. In particolare, se un oggetto *x* appartiene ad una classe che implementa *Iterable<A>*, per una data classe *A*, è possibile scrivere il seguente ciclo: `for (A a: x) {/* corpo */}` che è **equivalente** al blocco seguente:

```
Iterator<A> it = x.iterator();
while (it.hasNext()) {
    A a = it.next();
    /* corpo */
}
```

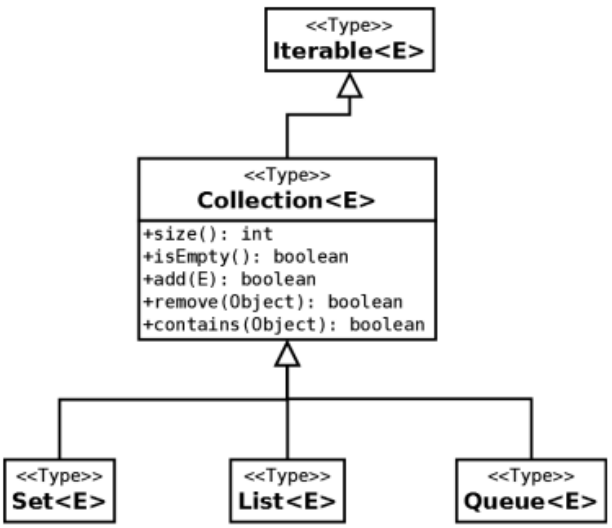
Come si vede, il ciclo enhanced-for è più sintetico e riduce drasticamente il rischio di scrivere codice errato.

Type checking del for-each: il seguente ciclo:
`for (A a: <exp>) {/*corpo*/}` è corretto a queste condizioni:

1) *<exp>* è una espressione di tipo (dichiarato) array di *T* oppure di un sottotipo di *Iterable<T>*
2) *T* è assegnabile ad *A*

L'interfaccia Collection

Il diagramma a destra illustra le interfacce direttamente collegate con ***Collection***; abbiamo già detto che *Collection* estende *Iterable*, e quindi fornisce un metodo che restituisce un iteratore. Come si vede anche dallo schema, *Collection* viene estesa dalle seguenti interfacce parametriche:



- ***Set*** rappresenta un insieme in senso matematico
  - Non sono ammessi duplicati
  - L'ordine in cui gli elementi vengono inseriti non è rilevante
- ***List*** rappresenta un vettore
  - sono ammessi duplicati
  - gli elementi vengono mantenuti nello stesso ordine in cui sono stati inseriti
- ***Queue*** rappresenta una coda

Come dice il nome, una Collection rappresenta un insieme di oggetti. Il tipo degli oggetti contenuti è indicato tramite un parametro di tipo presente nell'interfaccia.
I metodi principali dell'interfaccia *Collection* sono i seguenti:

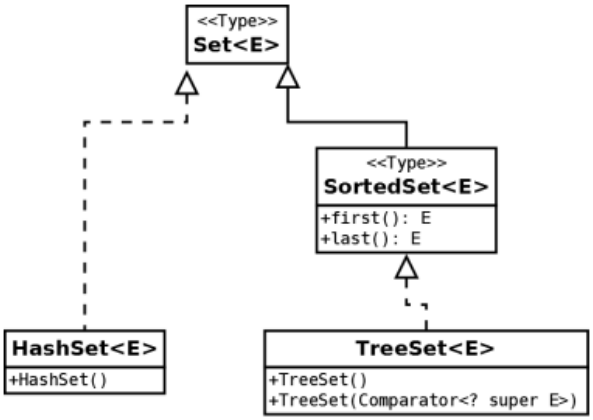
- ***int size()***: restituisce il numero di oggetti contenuti
- ***boolean isEmpty()***: restituisce vero se e solo se la collezione è vuota
- ***boolean add(E x)***: aggiunge *x* alla collezione, se possibile; restituisce vero se e solo se *x* è stato aggiunto con successo
- ***boolean contains(Object x)***: restituisce vero se e solo se la collezione contiene un oggetto uguale (nel senso di equals) ad *x*
- ***boolean remove(Object x)***: rimuove l'oggetto *x* (o un oggetto uguale ad *x* secondo equals) dalla collezione; restituisce vero se e solo se un tale elemento era presente nella collezione

Come si è visto, le collezioni fanno affidamento al metodo **equals** per identificare gli elementi ma le implementazioni concrete di *Collection* usano anche altri metodi (vedremo in seguito). Per quanto riguarda il metodo *add*, il motivo per cui restituisce un valore booleano diventerà chiaro quando esamineremo alcune implementazioni concrete di *Collection*.

Può sorprendere che i metodi *contains* e *remove* accettino *Object* invece del tipo parametrico *E*, ma lo fanno perché non si corre alcun rischio a passare a questi due metodi un oggetto di tipo sbagliato poiché, semplicemente, entrambi i metodi restituiranno false, senza nessun effetto sulla collezione stessa.
Inoltre, in certi casi la possibilità di passare oggetti qualsiasi potrebbe tornare utile; ad esempio, potremmo trovarci in un metodo che ha a disposizione una *Collection<String>* e che riceve dall'esterno un Object, che potrebbe essere di tipo effettivo stringa e la firma di *contains* consente di passargli quest'oggetto, senza doverci preoccupare preventivamente di controllare se si tratta di una stringa o meno.

L'interfaccia Set e le sue implementazioni

L'interfaccia *Set* non aggiunge metodi a *Collection*; tuttavia, restringe, ovvero rende più specifici, i contratti di alcuni metodi, come *add*. La specificità di un *Set*, rispetto ad una *Collection* generica, è che un *Set* **non può contenere elementi duplicati**; quindi, se si tenta di aggiungere con *add* un elemento che è già presenta (ovvero, un oggetto che risulta uguale, secondo equals, ad uno già presente), la collezione non viene modificata e add restituisce **false**.



SortedSet

L'interfaccia ***SortedSet***, che estende *Set*, rappresenta un insieme sui cui elementi è definita una relazione d'odine (totale).
L'iteratore di un *SortedSet* garantisce che gli elementi saranno visitati in ordine, dal più piccolo al più grande; inoltre, un tale insieme dispone di due metodi extra che non modificano la collezione:

- ***first*** restituisce l'elemento minimo tra quelli presenti nella collezione
- ***last*** restituisce l'elemento massimo

TreeSet

TreeSet è un insieme, implementato internamente come **albero di ricerca bilanciato**. Ne consegue che gli elementi devono essere dotati di una relazione d'ordine, in uno dei seguenti modi:

- Gli elementi sono dotati di ordinamento naturale; in questo caso, si può utilizzare il costruttore di *TreeSet()* senza argomenti.
- Gli elementi sono dotati di un ordinamento speciale, in questo caso si usa il costruttore *<E> TreeSet(Comparator<? super E>)* passandogli un opportuno oggetto Comparator;

Si potrebbe pensare di dichiarare *TreeSet<E extends Comparable>* ma così escluderemo tutte le classi che non estendono *Comparable* anche se hanno un ordinamento implementato da *Comparator*. Questo è il motivo per cui la dichiarazione *TreeSet<E>* è quella usata anche se, a causa dell'erasure, mi accorgo che la mia classe non abbia un ordinamento (con *Comparable* o *Comparator*) solo con un implementazione concreta; quindi, praticamente avrei errore solo alla prima *add*.

Le operazioni principali di *TreeSet* hanno la seguente complessità di tempo, tipica degli alberi di ricerca bilanciati: $O(\log n)$ per *add*, *contains* e *remove*; mentre, $O(1)$ per *size*, *isEmpty*, *first* e *last*.

TreeSet utilizza un ordinamento, fornito tramite *Comparable* o da *Comparator*, per **smistare** e poi **ritrovare** gli elementi all'interno dell'albero; in particolare, **se due oggetti sono equivalenti per l'ordinamento, saranno considerati uguali dal *TreeSet***.

Allo stesso tempo, l'interfaccia *Set* prevede che si usi equals per identificare gli elementi; quindi, se l'ordinamento non è coerente con l'uguaglianza definita da equals, *TreeSet* può **violare** il contratto di *Set*. Ad esempio, il codice seguente ha un comportamento differente da quello che ci si aspetterebbe con *Set*:

```
SortedSet<String> s = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
add("uno"); // restituisce true
add("UNO"); // restituisce false poiché fedele all'ordinamento
```

Se vogliamo un *TreeSet* che rispetti il contratto di *Set*, dobbiamo fornire un ordinamento **coerente con equals**. Consideriamo il caso di *Comparable*, per ogni coppia di elementi *x* e *y*, in aggiunta alle normali proprietà di equals e compareTo, deve valere la seguente condizione: *x.equals(y)* è vero se e solo se *x.compareTo(y) == 0*. Una condizione analoga deve valere nel caso venga fornito un oggetto di tipo C*omparator*.

HashSet

HashSet è un *Set* realizzato internamento come **tabella hash**. Utilizza il metodo *hashCode* della classe Object per selezionare il bucket in cui posizionare un elemento: *public int hashCode()*
I principali metodi di *HashSet* hanno complessità media costante ($O(1)$), ma le prestazioni reali dipendono dalla "bontà" della funzione hash utilizzata (più gli oggetti vengono sparsi per tutti i bucket meglio è).

Come previsto dall'interfaccia *Set*, *HashSet* utilizza il metodo equals per identificare gli elementi; quindi, equals ed hashCode devono rispettare la seguente **regola di coerenza**: per ogni coppia di elementi *x* ed *y* se *x.equals(y)* è vero allora *x.hashCode() == y.hashCode()*.
Si noti che le versioni di equals ed hashCode presenti in Object rispettano tale regola; infatti, se *x.equals(y)* è vero, allora *x* ed *y* puntano al medesimo oggetto, e quindi *x* ed *y* hanno lo stesso hashCode.

Ridefinizione di hashCode

Per capire l'importanza della coerenza tra equals ed hashCode esaminiamo il caso di una classe che non rispetta la regola di coerenza tra equals ed hashCode.
Supponiamo che una classe *Employee e* ridefinisca equals in modo che confronti il nome degli impiegati, ma non ridefinisca hashCode di conseguenza; quindi, due oggetti *Employee x* ed *y* con lo stesso nome risulteranno uguali secondo equals, ma avranno hash diversi. Se inseriamo l'oggetto *x* in un *HashSet*: `Set<Employee> s = new HashSet<Employee>();  s.add(x);` una successiva chiamata a *s.contains(y)* **potrebbe restituire false**; infatti, il codice hash di *y* potrebbe indurre la struttura dati a cercare l'oggetto in questione in un bucket diverso da quello in cui è stato inserito *x*.

Riassumendo, una buona ridefinizione di *hashCode* deve rispettare le seguenti proprietà:

1) **Coerenza con equals** (necessario): $\forall x, y$ se *x.equals(y)* è vero allora *x.hashCode() == y.hashCode()*
2) **Coerenza temporale** (necessario): il valore deve dipendere solo dallo stato dell'oggetto
3) **Uniformità** (desiderabile): il valore di ritorno deve essere uniformemente distribuito sugli interi

Perché *HashSet* funzioni in maniera efficiente, ed in particolare perché le operazioni principali abbiano complessità costante, è necessario che la classe componente (cioè, la classe degli elementi contenuti) disponga di un opportuno metodo hashCode. È importante che il metodo hashCode assegni ad ogni oggetto un numero intero il più possibile **uniformemente distribuito**.

Ad esempio, supponiamo che la classe *Employee* disponga dei campi *name* (String) e *salary* (int), e che siano considerati uguali gli impiegati che hanno questi due campi uguali.
Il metodo hashCode dovrebbero restituire un intero derivato dai valori dei due campi. Per i tipi non numerici, come le stringhe, conviene partire dal loro codice hash, poi, si devono combinare gli interi ottenuti dai vari campi in un unico numero, che verrà restituito. Un modo comune di combinare due interi, senza correre il rischio di andare in overflow, è rappresentato dall'or esclusivo (XOR) bit a bit, eseguito in Java dall'operatore bit a bit ^

Quindi, per l'esempio in questione, si potrebbe procedere come segue:

```
public int hashCode() { return name.hashCode() ^ salary }
```

Mutabilità degli elementi

Un problema che affligge sia *TreeSet* che *HashSet* riguarda la mutabilità degli elementi. Gli elementi vengono inseriti in queste strutture dati in base al valore dei loro campi:

- Un *TreeSet* posiziona gli elementi sulla base di confronti con altri oggetti
- Un *HashSet* li posiziona sulla base del loro codice hash

Se un elemento viene modificato dopo essere stato inserito in una di queste strutture dati, il posizionamento di quell'elemento non corrisponderà più al valore dei suoi campi; in questo caso, la struttura dati si comporterà in modo inatteso. Ne è un esempio il seguente codice:

```
Set<Employee> s = new HashSet<>();
Employee e = new Employee("Pippo", 1200);
s.add(e);
e.setSalary(1300);
s.contains(e); // risultato imprevedibile
```

Supponendo che *Employee* ridefinisca *hashCode* in modo che operi sul salario (come nell'esempio precedente). Situazione analoga può capitare nei *TreeSet* (ad esempio, ordino per salario e lo cambio in seguito). Quindi, la giusta procedura è quella di togliere l'oggetto dalla collezione, modificarlo e rimetterlo successivamente. Ma è evidente che questo processo risulta molto dispendioso e per questo si preferisce utilizzare in questo tipo di collezioni solo oggetti immutabili (nel caso si utilizzino oggetti mutabili bisogna segnalarlo nel contratto della classe per informare gli utenti).

L'interfaccia List e le sue implementazioni

List rappresenta una sequenza di elementi, ovvero un vettore. La figura a destra rappresenta l'interfaccia *List* e le sue implementazione; a differenza di *Set*, l'interfaccia *List* aggiunge alcuni metodi a *Collection*.

Presentiamo solo i due metodi responsabili per l'**accesso posizionale** e in seguito analizzeremo le due principali classi che implementano *List*: *LinkedList ed ArrayList.*

- ***get(int i)*** restituisce l'elemento al posto i-esimo della sequenza; solleva un'eccezione se l'indice è minore di zero o maggiore o uguale di *size()*;
- ***set(int i, E elem)*** sostituisce l'elemento al posto i-esimo della sequenza con *elem*; restituisce l'elemento sostituito; come *get*, solleva un'eccezione se l'indice è scorretto.
  Nota bene: non si può usare *set* per allungare una lista.

LinkedList

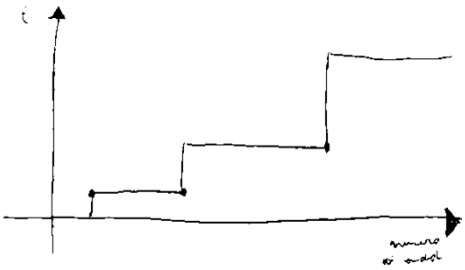
La classe ***LinkedList*** offre, in aggiunta a quelli di *List*, i seguenti metodi:

`public void addFirst(E elem)`	aggiunge x in testa alla lista
`public void addLast(E elem)`	equivalente ad add(x), ma senza valore restituito
`public E removeFirst()`	rimuove e restituisce la testa della lista
`public E removeLast()`	rimuove e restituisce la coda della lista

- Questi metodi permettono di utilizzare una *LinkedList* sia come stack sia come coda
- Per ottenere il comportamento di uno **stack** (detto **LIFO**: last in first out), inseriremo ed estrarremo gli elementi dalla stessa estremità della lista; ad esempio, inserendo con *addLast* (o con *add*) ed estraendo con *removeLast*
- Per ottenere, invece, il comportamento di una **coda** (**FIFO**: first in first out), inseriremo ed estrarremo gli elementi da due estremità opposte

ArrayList

ArrayList è un'implementazione di *List*, realizzata internamente con un array di dimensione dinamica; ovvero, quando l'array sottostante è pieno, esso viene riallocato con una dimensione maggiore, e i vecchi dati vengono copiati nel nuovo array. Questa operazione avviene in modo non visibile per l'utente ed il metodo *size* restituisce il numero degli elementi effettivamente presenti nella lista (non la dimensione dell'array sottostante).



Il ridimensionamento avviene in modo che l'operazione di inserimento (*add*) abbia **complessità ammortizzata costante**:

- I picchi nel grafico avvengono quando bisogna creare un nuovo array e copiare gli elementi precedenti. Ovviamente i picchi aumentano di intensità per ogni nuovo array creato ma avvengono ad intervalli sempre più lunghi.
- La complessità è ammortizzata proprio per quest'ultimo motivo. Posto $T(n) = \text{costo di } n \text{ add}$ allora la complessità ammortizzata sarà $\frac{T(n)}{n}$ (media temporale delle sequenza)
- Supponiamo che la capacità iniziale $init$ sia 10 e che il fattore di crescita $grow$ sia 1.5 (la dimensione del vettore aumenta del 50%); sviluppiamo $T(n)$

$$T(n) = \underbrace{10}_{init} + \underbrace{1 + \cdots + 1}_{10\ add} + \underbrace{15}_{grow} + \underbrace{1 + \cdots + 1}_{5\ add} + \underbrace{22}_{grow} + \underbrace{1 + \cdots + 1}_{7\ add} + \cdots = \underbrace{n}_{\substack{somma\\di\ tutti\\gli\ add}} + \underbrace{\sum_{i=0}^{k} 10 \cdot (1{,}5)^i}_{init+i\ grow}$$

dove k è il più piccolo intero che soddisfa la seguente disequazione: $10(1{,}5)^k \geq n$ e quindi risulta $k \geq \log_{1,5}\left(\frac{n}{10}\right)$; ovvero, $k = \left\lceil \log_{1,5}\left(\frac{n}{10}\right) \right\rceil$. Per semplicità consideriamo questo k un intero proprio uguale ad $\log_{1,5}\left(\frac{n}{10}\right)$:

$$T(n) = n + 10 \sum_{i=0}^{\log_{1,5}\left(\frac{n}{10}\right)} (1{,}5)^i = n + 10\left(\frac{(1{,}5)^{\log_{1,5}\left(\frac{n}{10}\right)+1} - 1}{1{,}5 - 1}\right) = n + 10\,\frac{1{,}5 \cdot 1{,}5^{\log_{1,5}\left(\frac{n}{10}\right)} - 1}{\frac{1}{2}}$$

$$= n + 20\left(\frac{3}{2} \cdot \frac{n}{10} - 1\right) \text{ essendo } 1{,}5^{\log_{1,5}\left(\frac{n}{10}\right)} = \frac{n}{10}$$

dunque, $T(n) = n + 3n - 20 = 4n - 20 = \mathrm{O(n)}$; e quindi, nonostante i picchi crescano di intensità, la complessità rimane lineare nel tempo. Di conseguenza la complessità ammortizzata è praticamente costante.

L'accesso posizionale (metodi *get* e *set*) si comporta in maniera molto diversa in *LinkedList* rispetto ad *ArrayList*:

- In *LinkedList*, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista (complessità lineare); difatti, per accedere all'elemento di posto n è necessario scorrere la lista, a partire dalla testa, o dalla coda, fino a raggiungere la posizione desiderata
- In *ArrayList*, ogni operazione di accesso posizionale richiede tempo costante; pertanto, se l'applicazione richiede l'accesso posizionale, è opportuno utilizzare *ArrayList* (fortemente sconsigliato *LinkedList*).

Il fatto che l'accesso posizionale sia indicato per *ArrayList* e sconsigliato per *LinkedList* è anche segnalato dal fatto che, delle due, solo *ArrayList* implementa l'interfaccia ***RandomAccess***, la quale è un interfaccia "di tag", come *Cloneable*; ovvero, è un interfaccia vuota che serve solo a segnalare che la classe che la implementa offre l'accesso posizionale in maniera efficiente (*get* e *set* sono a tempo costante). Praticamente rappresenta una modifica di contratto per i metodi *get* e *set*, poiché la proprietà astratta sopradescritta non è scrivibile nel linguaggio (per questo *RandomAccess* non ha metodi o variabili).

Avere un interfaccia di tag può essere utile se si vuole creare un metodo il più efficiente possibile, di seguito ne proponiamo un esempio (in base alla lista passata usiamo l'algoritmo più efficiente per invertirla):

```
static <T> void reverse(List<T> l) {
    if (l istanceof RandomAccess) {
        for (int i = 0; i < l.size()/2; ++i) {
            T temp = l.get(i);
            l.set(i, l.get(l.size() - i - 1));
            l.set(l.size() - i - 1, temp);
        }
    }
    else {
        List<T> temp = new ArrayList<>(l.size());
        Iterator<T> i = l.iterator();
        while (i.hasNext()) {
            temp.add(i.next());
            i.remove(); // uso la funzione dell'iteratore poiché usare
        }               // il remove di lista mi darebbe errore
        for (int j = temp.size() - 1; j >= 0; --j)
            l.add(temp.get(j));
    }
}
```

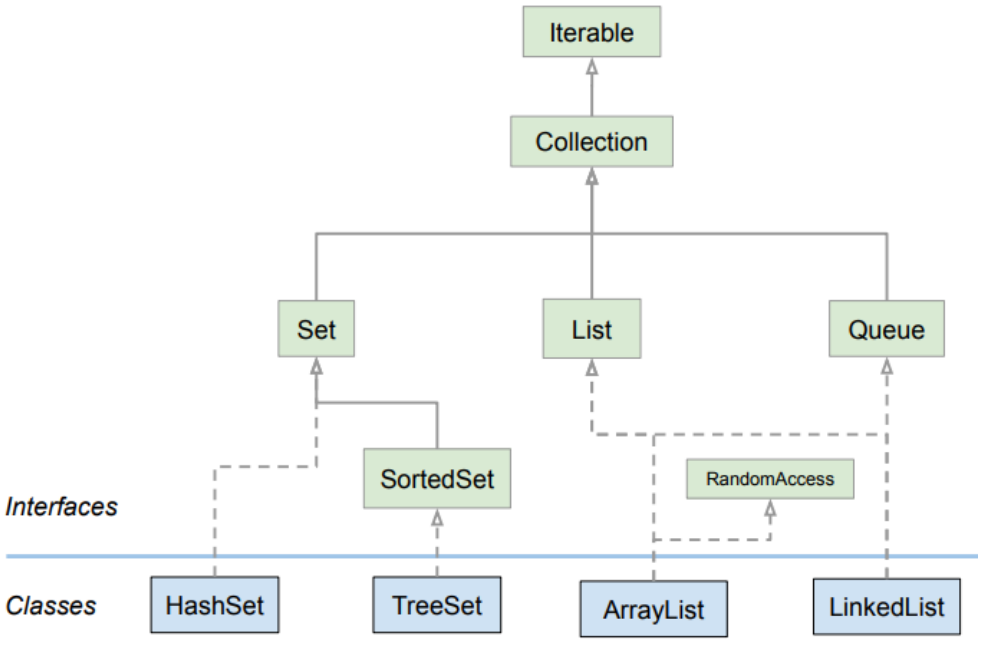
Confronto tra LinkedList e ArrayList

La seguente tabella riassume la complessità computazionale dei principali metodi delle liste:

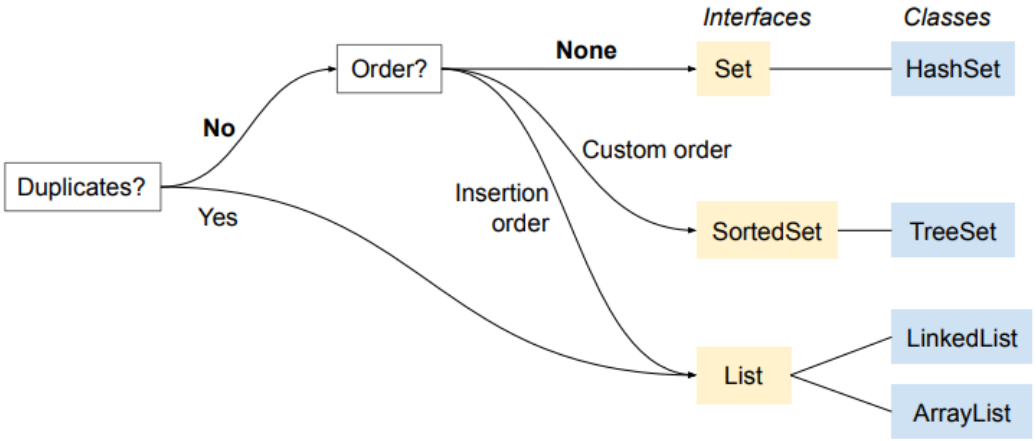
Metodo	Complessità in LinkedList	Complessità in ArrayList
add	$O(1)$	$O(1)$ (complessità ammortizzata)
remove	$O(n)$	$O(n)$
contains	$O(n)$	$O(n)$
get, set	$O(n)$	$O(1)$
addFirst, addLast, removeFirst, removeLast	$O(1)$	-

Note: *add* aggiunge in coda; *remove* deve trovare l'elemento prima di rimuoverlo

Collezioni: diagramma riassuntivo

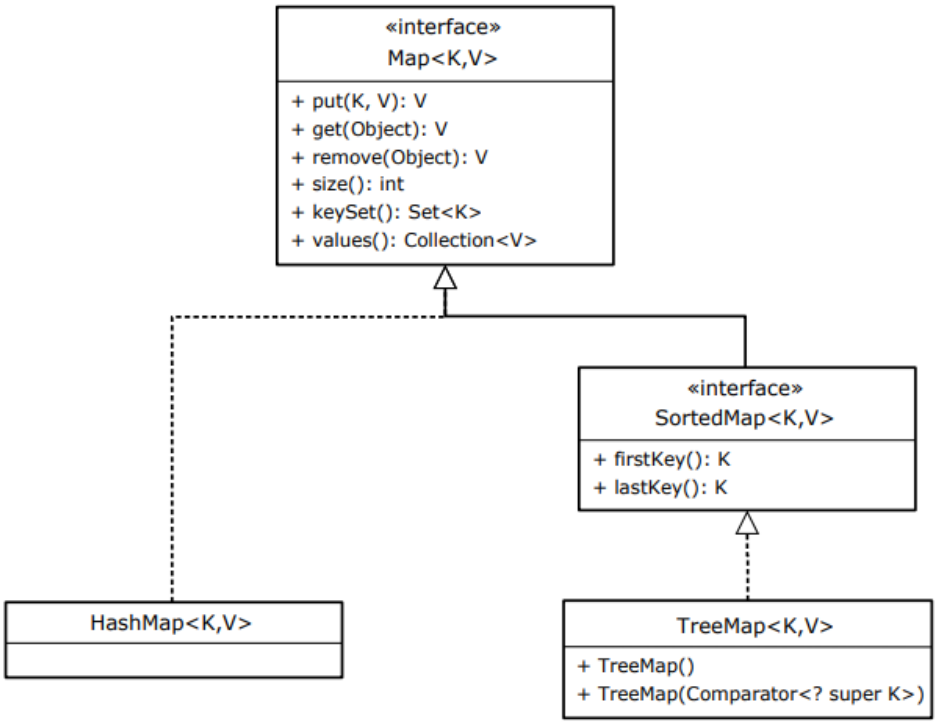


Il seguente schema sintetizza come scegliere una collezione:



L'interfaccia Map e le sue implementazioni

Per mappa si intende un'**insieme di coppie chiave-valore**, senza chiavi duplicate. Una mappa è analoga ad una funzione parziale da un insieme di chiavi ad uno di valori; in altri linguaggi, questo tipo di struttura dati prende il nome di **dizionario** o **array associativo** (nota che anche se si definisce array, a differenza di quest'ultimi, nelle mappe gli oggetti non sono inseriti in un area contigua di memoria).



Descriviamo i metodi principali dell'interfaccia *Map<K, V>*:

- ***V put(K k, V v)***: inserisce una nuova coppia e restituisce il valore precedentemente associata alla chiave *K*, se la chiave inserita è diversa da quelle presenti allora restituisce null.
- ***V get(Object o)***: restituisce il valore associato alla chiave passata come parametro (null in caso la chiave non esista). Il fatto che accetti Object è un po' un eccesso di generalizzazione, è una scelta di Java (discutibile) ma andrebbe benissimo anche un *V get(K k)*.
- ***Set<K> keySet()***: restituisce la mappa sotto forma di un insieme di chiavi (si usa *Set* poiché le chiavi non ammettono duplicati)
- ***Collection<V> values()***: restituisce la mappa sotto forma di un insieme di valori (non si usa *List* poiché non è previsto un ordinamento nelle coppie, quindi, è più generico *Collection*)

Gli ultimi due metodi descritti vengono denominati **viste** poiché queste non occupano spazio aggiuntivo creando una nuova collezione, bensì cambia solo la presentazione dell'attuale collezione; quindi, fare qualcosa in questi insiemi è farlo alla mappa stessa. Questo è anche il motivo per cui non è possibile fare una *add* in *keySet()* (insensato inserire una chiave senza un valore) o in *values*.

Le viste sono assolutamente necessarie per iterare su una mappa poiché *Map* non implementa *Iterable*.
Il modo consigliato per iterare su *Map* è il seguente:

```
for (K key: map.KeySet()) {
    V val = map.get(key);
    // in questo modo abbiamo
    // tutte le coppie (key,val)
    // nella nostra Map<K, V> map
}
```

SortedMap richiede un ordinamento per le chiavi ed offre i due metodi aggiuntivi analoghi a *first* e *last* di *SortedSet*, in modo analogo *HashMap* sfrutta l'*hashCode* delle chiavi. Non discuteremo in dettaglio queste due implementazioni poiché sono praticamente analoghe ai corrispettivi *SortedSet* e *HashSet* viste precedentemente senza i valori (anzi si può dire che il codice usato di *HashSet* e *SortedSet* è lo stesso di quello usato rispettivamente da *SortedMap* e *HashMap* per le chiavi).

Forniamo un esempio sull'utilizzo delle mappe fornendo un metodo statico che accetta una *Map* e restituisce vero se la mappa è iniettiva (quindi a chiavi diverse corrispondono valori diversi):

- Si potrebbe pensare in un primo momento di scrivere semplicemente un *return map.KeySet().size() == map.values().size();* ma non c'è nulla di più sbagliato.
  Infatti, in questo modo si restituisce sempre true poiché *values* ritorna una collezione con tutti i valori, duplicati compresi e quindi il numero dei valori sarà sempre uguale a quello delle chiavi
- La soluzione è dunque quella di usare un Set di appoggio:

```
public static <K,V> boolean isInjective(Map<K,V> map) {
    Set<V> temp = new HashSet<>();
    for (V val: map.values())
        if(!temp.add(val))
            return false; // se arrivo qui ho trovato un valore duplicato
                          // e quindi la funzione non sarà iniettiva
    return true; // se il cilco finisce allora ho sicuramente #K = #V
}
```

- Si noti che non abbiamo mai usato il parametro *K*, di conseguenza non ha senso tenerlo come tipo parametrico; dunque, si potrebbe fare *<V> boolean isInjective(Map<?,V> map)*. Ma possiamo fare a meno anche del parametro *V*, e quindi possiamo risparmiarci di utilizzare una funzione parametrica (se la si può evitare tanto meglio).
- Il seguente codice è analogo al precedente (ma non è un metodo parametrico):

```
public static boolean isInjective(Map<?,?> map) {
    Set<Object> temp = new HashSet<>();
    for (Object val: map.values())
        if(!temp.add(val))
            return false;
    return true;
}
```

13. Enumerazioni

Il Typesafe Enum Pattern

Un'esigenza ricorrente nella programmazione consiste nel creare un tipo di dati che può assumere un numero **finito** e **limitato** ("piccolo") di valori (ad esempio i giorni della settimana o i semi delle carte).

Questo tipo di strutture è conosciuto come **enumerazione** e molti linguaggi di programmazione ne danno un supporto (anche se limitato) di questa struttura. Java inizialmente non aveva alcun supporto per queste strutture e quindi il programmatore doveva gestire il tipo di dato.

Proviamo a rappresentare i semi delle carte francesi. Una possibile implementazione potrebbe essere quella di usare valori interi come rappresentazione dei colori (*0* per "Hearts", *1* per "Spades", …) oppure usare delle stringhe("*Hearts*", "*Spades*", …). Ma entrambe queste soluzioni hanno dei problemi:

- Con gli interi il codice risulta poco chiaro mentre con le stringhe abbiamo ovviamente una complessità maggiore per le operazioni
- Entrambe inoltre risultano poco rappresentative: per "Hearts" uso *1* o *0*? Per le stringhe uso "*Hearts*", "*HEARTS*" o "*hearts*"?

Delle due soluzioni è comunque preferibile la rappresentazioni tramite interi poiché ho almeno un ordinamento tra gli elementi.

Ma la soluzione migliore è usare una classe. Ne forniamo una d'esempio usando sempre i semi delle carte:

```
public class SuitClass {
    private String name;
    private int ord; // diciamo che vogliamo fornire un ordine tra i semi

    private SuitClass(String name, int ord) {
    this.name = name; this.ord = ord;
    }

    // i nostri elementi saranno ovviamente attributi statici
    public static final SuitClass HEARTS = new SuitClass("Hearts", 0);
    public static final SuitClass SPADES = new SuitClass("Spades", 1);
    public static final SuitClass CLUBS = new SuitClass("Clubs", 2);
    public static final SuitClass DIAMONDS = new SuitClass("Diamonds", 3);

    // funzione che converte il seme in un intero
    public int ordinal() { return ord; }
    // funzione che converte il seme in una stringa
    public String getName() { return name; }

    // Utile potrebbe essere restituisce una collezione degli elementi
    public static final Set<SuitClass> All;
    static {
        HashSet<SuitClass> temp = new HashSet<>();
        temp.add(HEARTS); temp.add(SPADES);
        temp.add(CLUBS); temp.add(DIAMONDS);
        All = Collections.unmodifiableSet(temp);
        // così da rendere l'insieme NON modificabile dall'utente
    }
}
```

Questa classe è la soluzione nota come **typesafe enum pattern**, ovvero un modo sicuro per descrivere le enumerazioni nei linguaggi che non forniscono un supporto nativo.

Riassumendo, le regole da eseguire per il typesafe enum pattern sono le seguenti:

- La classe avrà **solo costruttori privati**
- Ogni possibile valore del tipo enumerato corrisponderà ad una **costante** pubblica di **classe**, cioè un campo *public static final*
- Ciascuna di queste costanti viene inizializzata usando uno dei costruttori privati

Le classi enumerate in Java

A partire dalla versione 1.5 di Java, il linguaggio offre supporto nativo ai tipi enumerati, tramite il concetto di **classe enumerate** che è un tipo particolare di classe, introdotta dalla parola chiave ***enum***, che prevede un numero fisso e predeterminato di istanze. Riprendendo l'esempio dei semi, la corrispondente classe enumerata può essere definita come segue:

```
public enum SuitEnum { HEARTS, SPADES, CLUBS, DIAMONDS; }
```

Una classe enumerata può consistere semplicemente di un elenco di valori possibili. Per molti versi, *SuitEnum* si comporta come *SuitClass* descritta in precedenza; in particolare, i quattro valori dichiarati si utilizzano proprio come costanti pubbliche di classe, come ad esempio *SuitEnum s = SuitEnum.HEARTS;* Però *SuitEnum* dispone di **molte più funzionalità** di *SuitClass*, offerte in modo automatico e trasparente per il programmatore. Si noti che a differenza di *SuitClass*, l'**ordine** in cui i valori sono definiti in *SuitEnum* **è significativo** (HEARTS sarà 0, SPADES 1, etc…)

La classe enumerata *SuitEnum* gode delle seguenti funzionalità implicite (built-in):

```
SuitEnum x = SuitEnum.DIAMONDS;
int i = x.ordinal();
String name = x.name();

// restituisce l'array di tutti i valori
SuitEnum[] allSuits = SuitEnum.values();

//metodo statico parametrico che restituisce il valore con quel nome
SuitEnum y = Enum.<SuitEnum>valueOf(SuitEnum.class, "HEARTS");
```

Ma una classe enumerata può essere molto più ricca di così; può contenere **campi**, **metodi** e **costruttori** come una classe normale con l'unica restrizione che riguardano i costruttori: tutti i costruttori devono avere visibilità **privata** o **default** e non è possibile invocarli esplicitamente con *new*, neanche all'interno della classe stessa.

Supponiamo, ad esempio, di voler distinguere i semi rappresentati sulle carte con il colore rosso (cuori e quadri) da quelli rappresentati in nero (fiori e picche). La nostra classe diventa dunque:

```
public enum SuitEnum {
    HEARTS(true), SPADES(false), CLUBS(false), DIAMONDS(true);
    private final boolean red;
    private SuitEnum(boolean red) {
        this.red = red;
    }
    public boolean isRed() {
        return red;
    }
}
```

Se una classe enumerata ha più costruttori, ciascun valore può essere costruito con un costruttore diverso. Inoltre, valgono le seguenti proprietà:

- In una classe enumerata, la **prima riga** deve contenere l'elenco dei valori possibili
- Le classi enumerate estendono automaticamente la classe parametrica **Enum**; quindi, le classi enumerate non possono estendere altre classi, più precisamente, ogni classe enumerata *E* estende *Enum<E>*
- Le classi enumerate sono automaticamente ***final***

Ordinali e nomi dei valori enumerati

Ad ogni valore di una classe enumerata è associato un **numero intero**, chiamato **ordinale**, che rappresenta il suo posto nella sequenza dei valori, a partire da zero (nel caso di *SuitEnum*, a *HEARTS* è associato *0*, a *Spades 1*, e così via).

Per passare **da valore enumerato a ordinale**, si usa il seguente metodo pubblico della classe *Enum* (quindi viene ereditato da tutte le classi enumerate): *public int* ***ordinal()***

Per l'**operazione inversa** (da intero a valore enumerato), si usa il seguente metodo statico, che ogni classe enumerata *E* possiede automaticamente (non appartiene alla classe *Enum*): *public static E[]* ***values()***. Questo metodo restituisce un array contenente tutti i possibili valori di *E*; quindi, per ottenere il valore di posto i-esimo, è sufficiente accedere all'elemento i-esimo dell'array restituito da *values*.

È anche possibile passare **da valore a stringa**, per farlo si usa il seguente metodo della classe *Enum* che restituisce il **nome** del valore enumerato: *public String* ***name()***.

Per il **passaggio inverso**, la classe *Enum* offre il seguente metodo statico parametrico:

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
```

- Restituisce il valore enumerato della classe *enumType* che ha nome *name*
- *<T extends Enum<T>>* richiede esplicitamente un dato di tipo enumerato
- *Class<T> enumType* è un attributo necessario per gestire l'erasure, poiché il primo parametro *T* serve solo a restituire il tipo esatto e quindi serve una informazione a runtime per "pescare" l'istanza giusta di quella enumerazione. Tale informazione è data proprio dall'attributo *enumType*
- Il parametro di tipo del metodo rappresenta la classe enumerata a cui lo si applica, esempio: *SuitEnum x = Enum.<SuitEnum>valueOf(SuitEnum.class, "HEARTS");*

Specializzazione dei valori enumerati

È possibile specializzare il comportamento di un valore enumerato rispetto agli altri valori della stessa enumerazione; in particolare, è possibile che un valore enumerato abbia una **versione particolare di un metodo** che è comune a tutta l'enumerazione.

L'esempio dei 4 semi divisi in rossi e neri si può realizzare anche nel modo seguente:

```
public enum SuitEnum {
    HEARTS {
        public boolean isRed() { return true; }
    },
    SPADES, CLUBS,
    DIAMONDS {
        public boolean isRed() { return true; }
    };
    public boolean isRed() { return false; }
}
```

- Per specializzare il comportamento di un valore, si inserisce il codice relativo subito dopo la dichiarazione di quel valore, racchiuso tra parentesi graffe
- Le versioni specializzate di *isRed* rappresentano un **overriding** del metodo presente in *SuitEnum*

Come ulteriore esempio, consideriamo una classe enumerata *BoolOp*, che rappresenta i principali operatori booleani binari (AND e OR). Vogliamo dotare la classe di un metodo *eval*, che accetta due valori booleani e calcola il risultato dell'operatore applicato a questi valori. Per farlo si sfrutta il fatto che **le enumerazioni possono avere metodi astratti** pur non essendo astratte esse stesse. È quindi possibile utilizzare la

seguente sintassi:

```
public enum BoolOp {
    AND {
        public boolean eval(boolean a, boolean b) { return a && b; }
    },
    OR {
        public boolean eval(boolean a, boolean b) { return a || b; }
    };
    public abstract boolean eval(boolean a, boolean b);
}
```

Specializzazione come classe anonima

Abbiamo già visto che un'enumerazione come: `public enum MyEnum { A, B, C; }` è molto simile alla seguente sintassi:

```
public class MyEnum {
    public static final MyEnum A = new MyEnum();
    ...
}
```

Ne segue che un valore specializzato, come ad esempio:

```
public enum MyEnum {
    A {
        // codice specifico per A
    }, B, C;
}
```

va interpretato alla stregua di:

```
public class MyEnum {
    public static final MyEnum A = new MyEnum() {
        // codice specifico per A
    };
    ...
}
```

Il codice specifico di un valore si comporta come se fosse inserito in una apposita **classe anonima**; infatti, nell'ambito del codice specifico di un certo valore è possibile inserire tutti i costrutti che possono comparire in una classe anonima, come **campi** e **metodi.**

Collezioni per tipi enumerati

Il Java Collection Framework offre delle collezioni specificamente progettate per le classi enumerate:

- ***EnumSet***, versione specializzata di Set
- ***EnumMap***, versione specializzata di Map

EnumSet e la sua implementazione

EnumSet è una classe che implementa *Set* ed è ottimizzata per contenere elementi di una classe enumerata, la sua intestazione completa è:

```
public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>
                                                 implements Cloneable, Serializable
```

- Il limite superiore del parametro di tipo impone che tale tipo estenda *Enum* di sé stesso
- Questo requisito è soddisfatto da tutte le classi enumerate

Internamente, un *EnumSet* è un **vettore di bit** (bit vector); ovvero, se un *EnumSet* dovrà contenere elementi di una classe enumerata che prevede *n* valori possibili, esso conterrà internamente un vettore di ***n* valori booleani** dove l'i-esimo valore booleano sarà vero se l'i-esimo valore enumerato appartiene all'insieme, e falso altrimenti (esempio, per rappresentare i giorni della settimana basta un vettore di 8 bit).

Questa rappresentazione permette di realizzare in modo estremamente efficiente (tempo costante) tutte le operazioni base sugli insiemi previste dall'interfaccia *Collection* (*add, remove, contains*).

È facile rendersi conto che questa tecnica implementativa non potrebbe funzionare su classi non enumerate, che non hanno un numero prefissato e limitato di valori possibili. Inoltre, implica che un *EnumSet* deve conoscere il numero di valori possibili che ospiterà.

Per motivi di performance la libreria offre due diverse versioni (sottoclassi di *EnumSet*): una per enumerazioni con 64 elementi o meno ed una per enumerazioni con più di 64 elementi. Pertanto, la classe *EnumSet* è **astratta** e non ha costruttori pubblici, per istanziarla si utilizzano dei metodi statici, chiamati **metodi factory**.

Uno dei metodi factory è il seguente:

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elemType)
```

- Questo metodo crea un *EnumSet* vuoto, predisposto per contenere elementi della classe enumerata *elemType*
- Passare l'oggetto di tipo *Class* corrispondente alla classe enumerata permette all'*EnumSet* di conoscere il numero di valori possibili che ospiterà
- Ad esempio: *Collection<SuitEnum> c = EnumSet.noneOf(SuitEnum.class);* (possiamo affidarci alla type inference per dedurre il parametro di tipo appropriato, cioè *E = SuitEnum*)

EnumMap e la sua implementazione

EnumMap è una classe che implementa *Map*, ed è ottimizzata per i casi in cui le chiavi appartengano ad una classe enumerata. Internamente, una *EnumMap* con valori di tipo *V* è semplicemente **un array di riferimenti di tipo *V*** dove l'ordinale delle chiavi funge da indice nell'array.

La sua intestazione completa è:

```
public class EnumMap<K extends Enum<K>,V> extends AbstractMap<K,V>
                                          implements Serializable, Cloneable
```

A differenza di *EnumSet*, *EnumMap* ha costruttori pubblici poiché non ci sono speciali ottimizzazioni per numeri piccoli e quindi non ha senso usare i metodi factory come in *EnumSet*.
Il più semplice costruttore di *EnumMap<K, V>* è il seguente:

```
public EnumMap(Class<K> keyType)
```

- Questo metodo crea una *EnumMap* vuota, predisposta per contenere chiavi della classe enumerata *keyType*
- Ad esempio: *Map<SuitEnum, Integer> m = new EnumMap<>(SuitEnum.class);*

14. La riflessione

La classe Class

La **riflessione** (conosciuta negli altri linguaggi come **introspezione**) è una caratteristica di Java che permette ai programmi di investigare a tempo di esecuzione sui tipi effettivi degli oggetti manipolati.

Il cardine della riflessione è rappresentato dalla **classe Class** dove ciascun oggetto di classe Class rappresenta una delle classi del programma:

- La JVM si occupa di istanziare un oggetto Class per ogni nuova classe caricata in memoria
- **Solo** la JVM può istanziare la classe Class
- Un oggetto di tipo Class contiene tutte le informazioni relative alla classe che esso rappresenta: i costruttori, i metodi, i campi, ed anche le eventuali classi interne.
- Tramite questo oggetto, è possibile conoscere a runtime le caratteristiche di una classe che non è nota al momento della compilazione

Persino i tipi primitivi, pur non essendo classi, hanno un corrispondente oggetto di tipo Class.

La classe Class ha un parametro di tipo *T* che rappresenta il tipo dell'oggetto rappresentato, più precisamente, il parametro di tipo di un oggetto Class indica il tipo che questo oggetto rappresenta. Ad esempio, se *x* è l'oggetto Class relativo alla classe *Employee*, il tipo di *x* è *Class<Employee>.*

Esaminiamo una piccola selezione dei metodi della classe ***Class<T>***:

- *public Class<?>* ***getClass()***: restituisce l'oggetto Class del tipo effettivo di *this*. È molto simile all'operatore .class (che può essere chiamato solo su attributi statici) ma ha natura completamente dinamica (studieremo in dettaglio il tipo di ritorno successivamente)
- *public String* ***getName()***: restituisce il nome di questa classe in formato stringa, completo di eventuali nomi di pacchetti
- *public T* ***newInstance()***: crea e restituisce un nuovo oggetto di questa classe, invocandone il costruttore senza argomenti che questa classe **deve** possedere, altrimenti lancia delle eccezioni
- *public static Class<?>****forName****(String name)*: restituisce l'oggetto Class corrispondente alla classe di nome *name*; la stringa *name* deve contenere anche l'indicazione degli eventuali pacchetti cui la classe appartiene
- *public Class<? super T>* ***getSuperclass()***: restituisce l'oggetto Class corrispondente alla superclasse diretta di questa; se questa è *Object*, oppure è un'interfaccia o un tipo base, il metodo restituisce null
- *public boolean* ***isInstance****(Object x)*: restituisce vero se, e solo se, il tipo effettivo di *x* è sottotipo di questa classe. È praticamente l'analogo di *x instanceof Type*, la differenza sta nel *Type*; infatti, mentre con l'operatore instanceof si richiede espressamente un tipo statico, con il metodo *isInstance* possiamo anche usare una espressione (*exp.isInstance(…)*). Ne consegue che questo metodo è più potente del corrispettivo operatore poiché è dinamico.

Ottenere riferimenti agli oggetti di tipo Class

Per ottenere un riferimento ad un oggetto di tipo Class, è possibile utilizzare tre tecniche:

1) Il metodo **getClass** della classe Object

- Come già detto in precedenza il metodo *public Class<?> getClass()* restituisce l'oggetto Class corrispondente al tipo **effettivo** di questo oggetto
- Studiamo in dettaglio il **tipo restituito** di *getClass*:
  - Supponiamo di applicare *getClass* ad un oggetto di tipo dichiarato *Employee*:

    ```
    Employee e = ...;
    ??? = e.getClass();
    ```

    A prima vista, ci aspettiamo di poter assegnare il risultato della chiamata ad una variabile di tipo *Class<Employee>* ma, se il tipo effettivo di *e* fosse *Manager*, l'oggetto restituito da getClass sarebbe di tipo *Class<Manager>*, che non è assegnabile a *Class<Employee>*.
  - Pertanto, il tipo più appropriato (cioè, più specifico) a cui vorremmo assegnare il risultato è *Class<? extends Employee>*, ma *Class<?>* (il tipo di ritorno di getClass) **non** è assegnabile a *Class<? extends Employee>*. Come fare?
  - Il tipo restituito di getClass dovrebbe esprimere il seguente concetto: "Applicato ad un'espressione di tipo dichiarato *A*, questo metodo restituisce un oggetto di tipo *Class*". Purtroppo, **non è possibile** esprimere in Java questa proprietà; ragion per cui, il type-checker tratta il metodo getClass **in modo particolare**, simulando il tipo restituito che il linguaggio non è in grado di esprimere (infatti, fino a Java 5 il tipo di ritorno era *Class<? extends Object>*, ma visto che in ogni caso c'è il problema descritto in precedenza, tanto vale avere semplicemente *Class<?>*)

2) L'operatore **.class**

- Tale operatore, essendo un operatore **unario postfisso**, si applica al nome di una classe o di un tipo primitivo, come in:

```
Employee.class
String.class
java.util.LinkedList.class
int.class
```

- L'operatore *.class* ha carattere **statico**, nel senso che il suo valore è noto al momento della compilazione quindi, applicato ad una classe *A*, forma un'espressione di tipo *Class<A>*. Ad esempio (notare che *c2* e *c3* sono due oggetti differenti):

```
Class<String> c1 = String.class;
Class<Integer> c2 = Integer.class;
Class<Integer> c3 = int.class;
```

3) Il metodo **forName** della classe Class

- La terza tecnica è rappresentata dal metodo statico forName della classe Class, che abbiamo già presentato precedenza.
- Questa tecnica ha carattere **dinamico**, in quanto il valore restituito da forName non è noto al momento della compilazione.

Per una migliore comprensione si riporta un esercizio svolto:
Data la dichiarazione: `Employee e = new Manager(...);` Per ognuna delle seguenti espressioni, dire se è corretta o meno, e in caso affermativo calcolarne il valore

1) `e instanceof Employee` — Vero
2) `e instanceof Manager` — Vero
3) `e.class instanceof Employee` — Errore in compilazione
   class è statico e quindi non posso passargli un'espressione o una variabile
4) `e.getClass() == Manager` — Errore in compilazione
   dopo l'== serve un'espressione e quindi non posso utilizzare *Manager* a caso
5) `e.getClass() == Employee.class` — Falso
6) `e.getClass() == "Manager"` — Errore in compilazione
   non è possibile richiamare == a due oggetti completamente diversi
7) `e.getClass() == Manager.class` — Vero
8) `e.getClass().equals(Manager.class)` — Vero (analogo alla 7)

Esempio di utilizzo della riflessione

Presentiamo un metodo che accetta un array e un oggetto Class e riempie l'array di nuove istanze della classe corrispondente

```
public static <T> void fill(T[] arr, Class<? extends T> c)
    throws InstantiationException, IllegalAccessException
{
    for (int i=0; i<arr.length; i++) {
        T x = c.newInstance();
        arr[i] = x;
    }
}
```

Il metodo *newInstance* può lanciare diverse eccezioni verificate, ad esempio nel caso in cui la classe in questione non possegga un costruttore senza argomenti, o se tale costruttore non sia accessibile.

Ecco una possibile invocazione:

```
Employee[] a = new Employee[10];
<Employee>fill(a, Manager.class);
```

Si noti il tipo di Class, che consente di istanziare oggetti di una sottoclasse del tipo dell'array.

Volendo, lo stesso metodo si potrebbe implementare senza il parametro di tipo, in questo caso la sintassi è la seguente:

```
public static void fill(Object[] arr, Class<?> c)
    throws InstantiationException, IllegalAccessException
{
    for (int i=0; i < arr.length; i++) {
        Object x = c.newInstance();
        arr[i] = x;
    }
}
```

Ma questa versione è meno potente, infatti nel caso in cui invocassimo *fill(a, String.class)*, con la variabile *a* dichiarata come array di *Employee* mi aspetterei un errore in compilazione (cosa che avrò con la versione parametrica), ma con questo metodo me ne accorgo solo a runtime.

Nota: purtroppo, si verifica un problema simile anche con la versione parametrica; infatti, la seguente invocazione è lecita: *<Object>fill(a, String.class)*.

Riflessioni vs generics

Consideriamo una classe per coppie di oggetti con i generics:

```
public class Pair<S> {
    private S first, second;
    public Pair(S a, S b) {
        first = a;
        second = b;
    }
    public void setFirst(S a) { first = a; }
    public S getFirst() { return first; }

    @Override
    public boolean equals(Object other) {
        if (!(other instanceof Pair))
            return false;
        Pair<?> p = (Pair) other;
        return first.equals(p.first) && second.equals(p.second);
    }
}
```

Siamo costretti a usare la classe grezza

Ora proviamo a simulare i generics con la riflessione:

```
public class Pair {
    private Object first, second;
    private final Class<?> type;
    public Pair(Class<?> c, Object a, Object b) {
        if (!c.isInstance(a) || !c.isInstance(b))
            throw new IllegalArgumentException();
        type = c;
        first = a;
        second = b;
    }
    public void setFirst(Object a) {
        if (!type.isInstance(a))
            throw new IllegalArgumentException();
        first = a;
    }
    public Object getFirst() { return first; }
```

Controlliamo a runtime quello che i generics controllano in fase di compilazione

```java
@Override
public boolean equals(Object other) {
    if (!(other instanceof Pair))
        return false;
    Pair<?> p = (Pair) other;
    if (p.type != type)
        return false;
    return first.equals(p.first) && second.equals(p.second);
}
```

Usiamo ancora la classe grezza

Però possiamo controllare il tipo effettivo di un'altra coppia

Attenzione: così una coppia di Manager risulta diversa da una coppia di Employee, anche se contengono gli stessi oggetti (due Manager)

Volendo si potrebbero combinare i due approcci:

```java
public class Pair<S> {
    private S first, second;
    private final Class<S> type;
    public Pair(Class<S> c, S a, S b) {
        type = c;
        first = a;
        second = b;
    }
    public void setFirst(S a) { first = a; }
    public S getFirst() { return first; }
    ...
}
```

Usare questa classe:
*Pair<String> p = new Pair<>(**String.class**, "uno, "due");*
è okay.
*Pair p = new Pair<>(**"pippo".getClass()**, "uno", "due");*
è un errore di compilazione poiché il tipo dichiarato di *"pippo.getClass()"* è *Class<? extends String>* e quindi incompatibile con *Class<String>*

Ovviamente di questi tre tipi di implementazione la migliore è la prima, le altre due sono state introdotte solo a scopo didattico (arduo trovare un caso implementativo in cui si preferisce una classe riflessiva).

Esplorare il contenuto di una classe

I metodi della classe *Class* permettono di ricavare numerose informazioni sulla classe in questione; in particolare, è possibile conoscere l'elenco di tutti i **campi**, **metodi** e **costruttori** appartenenti alla classe. Per ottenere tali informazioni, sono utili i seguenti metodi di *Class*:

```java
public Field[] getFields()         // Tutti i campi pubblici (anche ereditati)
public Field[] getDeclaredFields()// Tutti i campi dichiarati (anche privati)

public Method[] getMethods()
public Method[] getDeclaredMethods()

public Constructor[] getConstructors()
public Constructor[] getDeclaredConstructors()
```

- Il metodo *getFields* restituisce tutti i campi **pubblici** di questa classe, anche ereditati
  - *getMethods* è analogo a *getFields* ma per i metodi
  - *getConstructors* restituisce semplicemente i costruttori pubblici di questa classe
- Il metodo *getDeclaredFields* restituisce tutti i campi dichiarati in questa classe (e non nelle superclassi), **indipendentemente** dalla loro visibilità
  - Similmente, *getDeclaredMethods* e *getDeclaredConstructors*

La classe Field

La classe *Field* rappresenta un **campo** di una classe e dispone di metodi per leggere e modificare il contenuto di un campo, conoscere il suo nome e il suo tipo. In particolare, abbiamo:

*public String **getName**()*	restituisce il nome di questo campo

public Object ***get****(Object x)* *throws IllegalAccessException*	Restituisce il valore di questo campo nell'oggetto *x*. Se questo campo è di un tipo base, il suo valore viene racchiuso nel corrispondente tipo riferimento (ad esempio, *int* → *Integer*). Se questo campo è statico, il parametro x viene ignorato.
public void ***set****(Object x, Object val)* *throws IllegalAccessException*	Imposta a *val* il valore di questo campo nell'oggetto *x*. Se questo campo è statico, il parametro *x* viene ignorato
public Class<?> ***getType****()*	Restituisce il tipo di questo campo

Alcuni metodi sollevano l'eccezione verificata *IllegalAccessException*, quando si tenta di accedere ad un campo che non è accessibile a causa della sua visibilità

I metodi *get* e *set* di Field (così come altri metodi simili delle classi *Method* e *Constructor*) applicano le regole di visibilità previste dal linguaggio, cioè, lanciano l'eccezione verificata *IllegalAccessException* se si tenta di accedere ad un campo che non è visibile dalla classe in cui ci si trova.

È possibile disattivare questo controllo utilizzando i metodi della classe *AccessibleObject*, superclasse comune a *Field*, *Method* e *Constructor*.

La possibilità di aggirare i controlli è soggetta al Security Manager, l'oggetto che è responsabile dei permessi per le operazioni a rischio. Di default, la JVM si avvia senza un Security Manager, consentendo alle applicazioni di compiere qualsiasi operazione; invece, i browser che eseguono applet Java utilizzano dei Security Manager particolarmente restrittivi.

La classe Method

La classe *Method* rappresenta un **metodo** di una classe e dispone di metodi per conoscere il nome del metodo, il numero e tipo dei parametri formali e il tipo di ritorno; inoltre, è possibile invocare il metodo stesso. In particolare, abbiamo:

public String ***getName****()*	Restituisce il nome del metodo rappresentato.
public Object ***invoke****(Object x, Object...args)* *throws IllegalAccessException*	Invoca sull'oggetto *x* il metodo rappresentato, passandogli i parametri attuali *args*. Se il metodo rappresentato è statico, il parametro x viene ignorato. Restituisce il valore restituito dal metodo rappresentato

La sintassi *Object...args* indica che *invoke* accetta un numero variabile di argomenti (Metodo variadico)

Metodi variadici

Dalla versione 1.5, Java prevede un meccanismo per dichiarare **metodi con un numero variabile di argomenti** (metodi **variadici**, o, in breve, **varargs**).

Se *T* è un tipo di dati, con la scrittura `f(T ... x)` si indica un numero variabile di argomenti (anche zero), tutti di tipo *T*. I puntini sospensivi **devono** essere necessariamente tre.

Gli argomenti possono essere passati separatamente. Come in `f(x1, x2, x3)`, oppure tramite un array, come in `f(new T[] {x1, x2, x3})`. In entrambi i casi, all'interno del metodo *f*, si può accedere agli argomenti utilizzando *x* come un array di tipo *T*.

Ogni metodo può avere un solo argomento variadico, che deve essere l'**ultimo** della lista, come il metodo *invoke* della classe *Method*, illustrato precedentemente.

Riflessione in C++

Il C non fornisce alcun supporto alla riflessione poiché in C nessun oggetto conosce il proprio tipo, a differenza di Java dove tutti gli oggetti lo conoscono.

Il C++, invece, fornisce un supporto parziale alla riflessione, chiamato **RunTime Type Information** (RTTI).

Diremo che un oggetto "conosce il proprio tipo" se a partire dal suo indirizzo è possibile risalire all'identità del suo tipo; ovvero, se nel memory layout dell'oggetto è presente un puntatore o un identificativo della sua classe.

In C++, RTTI agisce solo su classi che hanno almeno un metodo virtuale (cioè sovrascrivibile). Gli oggetti di queste classi conoscono il proprio tipo, per permettere il binding dinamico, mentre, gli oggetti delle altre classi (che non hanno un metodo virtuale) non lo conoscono.

Operatore typeid: *typeid(exp)*

- Restituisce un oggetto di tipo *std::type_info* corrispondente al tipo effettivo di *exp*
- Non compila se il tipo dichiarato di *exp* non conosce il proprio tipo
- Simile a *getClass*

Dynamic cast: *dynamic_cast<type>(exp)*

- Un cast che controlla a runtime se *exp* è di tipo effettivo *type*
- Se non lo è, restituisce *nullptr* (in caso di tipo puntatore) o lancia un'eccezione (in caso di tipo riferimento)
- Non compila se il tipo dichiarato di *exp* non conosce il proprio tipo
- Simile a un cast Java tra riferimenti

15. Scegliere l'interfaccia di un metodo

Interfaccia, prototipo e firma

Per **interfaccia** di un metodo si intende la facciata che il metodo offre ai suoi chiamanti:

- Nome del metodo
- Lista dei parametri formali
- Tipo di ritorno
- Modificatori (visibilità, *static*, *final*, etc…)
- In Java, l'eventuale clausola "*throws…*"

Alcuni autori usano il termine method header per riferirsi all'interfaccia di un metodo. In C++, si usa il termine **prototipo** per indicare l'intestazione di un metodo o funzione.

Per **firma** (signature) di un metodo si intende invece il nome del metodo e la lista dei parametri formali (senza tipi di ritorno), la firma contiene le uniche informazioni che sono rilevanti ai fini del binding dinamico.

In questo capitolo discuteremo dei criteri da prendere in considerazione quando si voglia **scegliere oculatamente l'interfaccia migliore per un metodo** che deve svolgere un dato compito (contratto). L'argomento è particolarmente rilevante in alcuni contesti, tra cui: metodi che manipolano **collezioni** e metodi che appartengono a **librerie** destinate ad un ampio uso.

Le collezioni si presentato in una ricca gerarchia di classi ed interfacce parametriche, rendendo complessa la scelta dell'interfaccia migliore per un metodo.

Per il successo di una libreria, è fondamentale che le interfacce dei metodi, e più in generale l'interfaccia pubblica della libreria, sia ben progettata, in modo da essere utile in quanti più contesti è possibile.

Scelta dei parametri formali

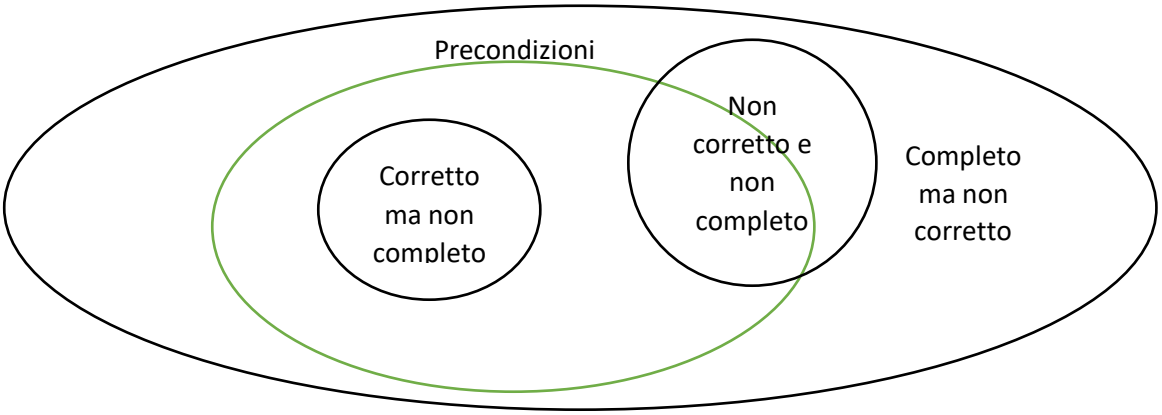
La scelta del tipo di parametri formali dovrebbe **rispecchiare il più fedelmente possibile la precondizione** del metodo in questione. Quindi, il tipo scelto dovrebbe accettare tutti i valori che soddisfano la precondizione (**completezza**) e rifiutare tutti gli altri valori (**correttezza**).

Purtroppo, spesso i limiti del linguaggio di programmazione usato impediscono di ottenere contemporaneamente correttezza e completezza.

Forniamo alcuni esempi sulla scelta dei parametri formali:

1) Il metodo accetta una **lista non vuota**
    - Non esiste in Java un tipo specifico per una lista non vuota
    - Ci accontenteremo di accettare una lista qualsiasi, per poi sollevare un'eccezione a runtime se la lista risulta vuota
    - La firma ottenute è completa ma non corretta
2) Il metodo accetta un **intero non negativo**
    - In Java non esiste un tipo specifico per un intero non negativo
    - Ci accontentiamo di accettare un intero qualsiasi, per poi sollevare un'eccezione se è negativo
    - La firma ottenuta è completa ma non corretta
    - In C++, sarebbe possibile ottenere una firma corretta e completa poiché esso ha a disposizione il tipo *unsigned int*

Prendiamo un insieme di precondizioni (praticamente gli elementi sono i valori validi), l'ideale sarebbe prendere esattamente quell'insieme per avere una firma corretta e completa. Possiamo approssimare gli altri casi come segue (l'insieme delle precondizioni è quello in verde):



Correttezza VS Completezza

Quando non esiste una scelta corretta e completa, si deve scegliere tra due opzioni:

A. Scartare tutti i valori non validi ed anche alcuni valori validi (firma corretta ma non completa)
B. Accettare tutti i valori validi ed anche alcuni valori non validi (firma completa ma non corretta)

Di norma, si preferisce la scelta B, in quanto i valori non validi che vengono accettati possono essere successivamente scartati a runtime (lanciando un'eccezione); quindi, in linea di massima assegneremo alla completezza una priorità più alta della correttezza.
In un contesto in cui sia prioritaria la robustezza e quindi si voglia minimizzare il rischio di errori a runtime, potrebbe essere preferibile la scelta A.

Il criterio di completezza o generalità richiede che il metodo accetti tutti i valori che soddisfano la precondizione. Ne consegue che **violare la completezza rende il metodo meno utile**, in quanto non è applicabile a tutti i valori previsti dal contratto; in altri termini, una firma non completa sta effettivamente **restringendo la precondizione**.

Invece, **violare la correttezza** comporta che il compilatore **consentirà ai chiamanti di passare al metodo in questione dei valori non validi**; in tal modo, si indebolisce la capacità del compilatore di verificare, tramite il type-checking, il rispetto delle precondizioni. La parte della precondizione che non viene espressa nella

firma dovrà essere verificata a runtime e potrà portare ad errori a tempo di esecuzione (lancio di eccezioni), così anche se la firma viola la correttezza, non si sta modificando (cioè indebolendo) la precondizione.

Consideriamo, da esempio, un metodo con la seguente **precondizione**: "accetta una collezione priva di duplicati". In Java, possiamo interpretare la precondizione in vari modi:

- `void f(Set<?>)`
  Corretta ma non completa. Anche una lista può essere priva di duplicati.
  Vantaggi: la precondizione è verificata dal compilatore. È impossibile per il chiamante sbagliare e violare la precondizione. Diminuisce il rischio di errori a tempo di esecuzione.
- `void f(Collection<?> s)`
  Completa ma non corretta. Il metodo può verificare a tempo di esecuzione che la collezione non contenga duplicati.
  Vantaggi: il metodo offre un servizio più ampio, cioè accetta anche le liste.

Altri criteri di valutazione

Per quanto sia comune scegliere una firma che accetta un tipo più generale di quello richiesto dalla precondizione, la firma non deve mai accettare un tipo così generico da impedire di portare a termine il compito assegnato al metodo. Quindi, è possibile scegliere una firma che violi la correttezza, a patto di preservare la **funzionalità** del metodo stesso; cioè, il tipo scelto deve contenere le informazioni (campi) e le funzionalità (metodi) necessarie a svolgere il compito previsto (nel valutare la funzionalità assumiamo di non voler ricorrere a conversioni forzate, ovvero ai cast).

In alcuni casi, è possibile esprimere nel tipo dei parametri formali delle **garanzie offerte dalla postcondizione**, come il fatto che un dato parametro non sarà modificato:

- In C++, il modificatore *const* serve proprio ad esprimere questo vincolo
  - Ad esempio, se *Person* è una classe, un parametro di tipo *const Person&* è un riferimento ad un oggetto *Person* che non può essere utilizzato per modificare l'oggetto
  - Tecnicamente, il riferimento in questione può essere usato solo per invocare metodi che siano a loro volta dichiarati *const*
- In java, il tipo jolly può esprimere una proprietà simile, nel caso delle collezioni
  - Un parametro di tipo *Collection<? extends Employee>* sostanzialmente non può essere utilizzato per modificare la collezione. Più precisamente, non è possibile invocare il metodo *add*, se non con argomento *null*; tuttavia, è possibile invocare il metodo *remove*, perché il suo argomento è di tipo *Object*
  - Analogamente, un parametro di tipo *Collection<? super Employee>* non può essere utilizzato per leggere il contenuto della collezione. Più precisamente, si può accedere agli oggetti contenuti soltanto come se fossero degli *Object*

Infine, un ultimo criterio che vale la pena menzionare riguarda la **semplicità dell'interfaccia**. A parità degli altri criteri, è preferibile un'interfaccia che sai più semplice da leggere e comprendere. Nel caso dei metodi parametrici, questo criterio implica che, a parità delle altre caratteristiche, è preferibile un'interfaccia che utilizza meno parametri di tipo.

Esempio addAll

Consideriamo un metodo *addAll* che accetta due collezione e aggiunge tutti gli elementi dalla prima alla seconda. Se andiamo a precisare questo contratto, otteniamo:

- **Precondizione**: accetta due collezioni, tali che gli elementi della prima possono essere inseriti nella seconda
- **Postcondizione**: alla fine dell'esecuzione, la seconda collezione conterrà gli elementi che conteneva all'inizio, più tutti quelli della prima collezione; la prima collezione non verrà modificata

Il fatto che gli elementi della prima collezione possano essere inseriti nella seconda si traduce in:
"Il tipo degli elementi della prima collezione è sottotipo del tipo degli elementi della seconda collezione"

Esaminiamo delle possibili interfacce per questo metodo:

1) `void addAll(Collection<?> a, Collection<?> b)`
   - È completa ma **non è corretta**, perché accetta dei valori come *a = Collection<String>* e *b = Collection<Employee>*, che non soddisfano la precondizione, perché non è possibile inserire delle stringhe in una collezione di Employee.
   - In questo caso, due problemi ci impediscono di accettare questa firma:
     - Per prima cosa, a causa delle limitazioni dei tipi generici in Java (si veda il capitolo sulle conseguenze dell'erasure), non sarebbe possibile verificare a tempo di esecuzione che le due collezioni fossero compatibili tra loro, perché i parametri effettivi di tipo delle due collezioni vengono persi durante la compilazione.
     - Cosa più importante, la firma in questione non consente di scrivere un corpo corretto per il metodo, in quanto **non è possibile invocare il metodo *add*** della collezione *b*!
   - Quindi, questa interfaccia **non è funzionale.**
2) `<T> void addAll(Collection<T> a, Collection<T> b)`
   - È funzionale, corretta ma **non completa**, perché non accetta valori validi come *a = Collection<Manager>* e *b = Collection<Employee>*
3) `<S, T extends S> void addAll(Collection<T> a, Collection<S> b)`
   - È funzionale, corretta e completa
4) `<T> void addAll(Collection<T> a, Collection<? super T> b)`
   - È funzionale, corretta e completa
   - È più semplice della 3 perché usa un solo parametro di tipo. Inoltre, garantisce che la collezione b non sarà letta.
5) `<T> void addAll(Collection<? extends T> a, Collection<T> b)`
   - È funzionale, corretta e completa
   - Usa un solo parametro di tipo e quindi è più semplice della 3. Inoltre, garantisce che la prima collezione non sarà modificata (almeno, non tramite il metodo add).
6) `<T> void addAll(Collection<? extends T> a, Collection<? super T> b)`
   - È funzionale, corretta e completa
   - Usa un solo parametro di tipo e garantisce che la prima collezione non sarà modificata e la seconda non sarà letta.

Quindi, scegliamo l'interfaccia 6. Infatti, nella classe Collections della API Java troviamo:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

Scelta dei parametri formali e del tipo di ritorno

Riassumendo, nella scelta del tipo di parametri formali si possono individuare le seguenti forze in gioco, elencate in ordine di **importanza decrescente**:

1) Funzionalità (il tipo prescelto deve offrire le funzionalità necessarie a svolgere il compito prefissato, cioè a realizzare la post-condizione)
2) Completezza (il metodo dovrebbe accettare tutti i valori che soddisfano la precondizione)
3) Correttezza (il metodo dovrebbe accettare soltanto valori che soddisfano la precondizione)
4) Ulteriori garanzie (il tipo dei parametri dovrebbe esprimere eventuali garanzie previste dalla postcondizione)
5) Semplicità (a parità degli altri criteri, la firma dovrebbe essere più semplice possibile)

Il criterio 1 è obbligatorio, mentre gli altri vanno considerati come qualità da massimizzare, ma che non sempre è possibile soddisfare a pieno, anche a causa delle limitazioni imposte dal linguaggio di programmazione utilizzato.

Praticamente (approssimando) una firma è tanto più buona quanti jolly ha (ovviamente se non messi a cazzo) e quanti meno parametri di tipo ha.

Nel caso del tipo di ritorno, solitamente l'unica decisione da prendere riguarda la sua specificità.
A questo proposito, due forze spingono in direzioni opposte:

1) Un tipo di ritorno **più specifico** è più utile per il chiamante, perché esprime più informazioni sul valore restituito
2) Un tipo di ritorno **meno specifico** nasconde l'implementazione interna del metodo (incapsulamento) e quindi favorisce l'evoluzione futura del software

Il progettista dovrà trovare un compromesso tra queste due forza e sarà il contesto a suggerire se è più importante l'utilità per il chiamante o la futura evoluzione del software in questione.

Di norma si utilizza l'interfaccia più specifica della classe utilizzata nel metodo come suo tipo di ritorno.
Best practice: Scegliere il tipo di ritorno in modo che conservi l'informazione di tipo presente nei parametri formali, ma nasconda i dettagli implementativi del metodo.
Ad esempio, tra *List<?>*, *List<Employee>* e *List<Manager>*, è più utile restituire *List<Manager>* perché fornisce più informazioni degli altri tipi sul valore restituito (questo vale anche se *List<Manager>* non è sottotipo di *List<Employee>*; se il chiamante non è interessato alle sottoclassi di *Employee*, può sempre assegnare il risultato a *List<? extends Employee>*). Invece, tra *List<Manager>* e *LinkedList<Manager>* di solito si preferisce la prima, per lasciare la possibilità di cambiare implementazione di lista in futuro (ad esempio, per poter usare ArrayList invece di LinkedList).

A volte, la specificità del tipo di ritorno può essere in contrasto con i criteri 4 e 5 (ulteriori garanzie e semplicità) della scelta dei parametri formali (come nell'esempio "intersezione insiemistica" nel seguito); in tal caso, solitamente si preferisce dare più importanza al tipo di ritorno.

Esempio: intersezione insiemistica

Consideriamo il problema di scegliere l'interfaccia di un metodo che accetta due insiemi (*Set*) e **restituisce l'intersezione dei due**. Prima di esaminare alcune interfacce possibili, è opportuno **chiarire il contratto** del metodo ed in particolare la sua precondizione.

Quali restrizioni devono valere sui due insiemi passati come argomenti? Idealmente, dovremmo trarre ispirazione dalla definizione matematica di intersezione, che definisce l'operazione su **qualsiasi coppia di insiemi**; quindi, se fosse possibile dovremmo accettare come argomenti due insiemi di **qualsiasi tipo**.

Può sorprendere che si vogliano accettare come argomenti anche due insiemi di tipi non "imparentati"; infatti, in alcuni casi, come un *Set<Employee>* e un *Set<String>*, sappiamo a priori che gli insiemi non possono contenere elementi in comune e quindi non ha senso calcolare l'intersezione, perché è sicuramente vuota, ne segue che potremmo scegliere di **scartare** questo tipo di argomenti.
Tuttavia, si consideri il caso di un *Set<Cloneable>* e un *Set<Comparable<?>>*, anche se le interfacce Cloneable e Comparable non hanno alcun collegamento, niente impedisce che tali insiemi abbiano degli elementi in comune (oggetti che implementano entrambe le interfacce); quindi, dovremmo accettare questo tipo di argomenti.

Consideriamo diverse interfacce:

1)
```
<T> Set<T> intersection(Set<T> a, Set<T> b)
```
È funzionale, corretta ma **non completa**. Rifiuta insiemi di tipo diverso.

2) `<T> Set<T> intersection(Set<T> a, Set<? extends T> b)`
È funzionale, corretta ma **non completa**. Accetta più della precedente ma comunque rifiuta insiemi di tipo non correlato, come *Set<Cloneable>* e *Set<Comparable<?>>*

3) `Set<?> intersection(Set<?> a, Set<?> b)`
È funzionale, completa e corretta. Già il solo fatto che sia completa la rende migliore delle precedenti ma è anche più semplice e esprime forti garanzie su entrambi i suoi argomenti.
Tuttavia, il suo difetto risiede nella poca specificità del tipo di ritorno; infatti, non conserva nessuna informazioni dei parametri formali.

4) `<S,T> Set<S> intersection(Set<S> a, Set<T> b)`
È funzionale, completa e corretta. Non esprime garanzie sugli argomenti ed è più complessa della precedente ma il tipo di ritorno è più specifico, quindi migliore, rispetto a quello della 3, in quanto conserva il tipo del primo argomenti.

5) `<S> Set<S> intersection(Set<S> a, Set<?> b)`
È funzionale, completa e corretta. È più semplice della 4 ma meno della 3, anche come garanzie si trova nel mezzo. Ciò rende questa interfaccia la scelta migliore di tutte le precedenti, in quanto è l'unica che:
- è funzionale, completa e corretta
- preserva nel tipo di ritorno una parte dell'informazione di tipo presente negli argomenti
- esprime la garanzia che il secondo argomento non verrà né letto né scritto
- usa un solo parametro di tipo

Si noti che è possibile scrivere *Set<? extends S> a* dando una piccola garanzia in più senza togliere nulla.

Per conferma, esaminiamo il metodo di intersezione fornito dalla libreria **Google Guava**, nella classe Sets:
```
public static <E> Sets.SetView<E> intersection(Set<E> a, Set<?> b)
```
Sets.SetView è una interfaccia che estende *Set* e rappresenta una vista su un insieme.

La struttura è proprio quella della nostra firma 5.

L'**API Java** offre invece il seguente metodo nell'interfaccia *Collection<E>*:
```
public boolean retainAll(Collection<?> other)
```
Il metodo modifica questa (*this*) collezione, lasciando solo gli elementi presenti nella collezione *other*. Quindi, **effettua l'intersezione tra *this* e *other***. Anche in questo caso, viene accettato un insieme di qualunque tipo.

16. Software Qualities

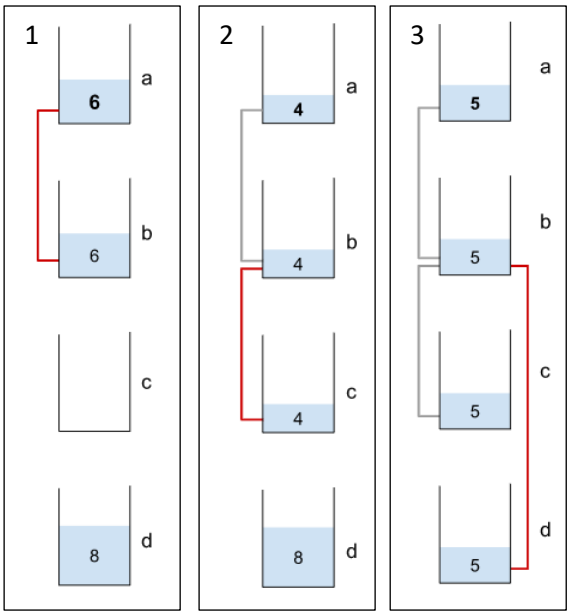
Un problema da risolvere

Supponiamo di voler implementare una classe ***Container***, che rappresenta un contenitore per liquidi di dimensione fissata. Il contenitore, inizialmente vuoto, dovrà avere i seguenti metodi:

- *double* ***getAmount****()*
  - restituisce la quantità d'acqua presente nel contenitore.
- *void* ***addWater****(double amount)*
  - prende come argomento il numero di litri ed aggiunge acqua al contenitore.
- *void* ***connectTo****(Container other)*
  - prende come argomento un altro contenitore, e lo collega a questo con un tubo. Dopo il collegamento, la quantità d'acqua nei due contenitori (e in tutti quelli ad essi collegati) sarà la stessa.

Per una migliore comprensione forniamo il seguente caso d'uso:

```
Container a = new Container(),
          b = new Container(),
          c = new Container(),
          d = new Container(),
// Caso 1
a.addWater(12);
d.addWater(8);
a.connectTo(b);
System.out.println(a.getAmount());
                  // Output: 6
// Caso 2
b.connectTo(c);
System.out.println(a.getAmount());
                  // Output: 4
// Caso 3
b.connectTo(d);
System.out.println(a.getAmount());
                  // Output: 5
```



Implementazione di riferimento (Reference)

Una possibile implementazione, che chiameremo **Reference**, è la seguente:

```
public class Container {
    private double amount;
    private Set<Container> group;

    public Container() {
        group = new HashSet<Container>();
        group.add(this);
    }

    double getAmount() { return amount; }

    void addWater(double amount) {
        double distributed = amount / group.size();
        for (Container c: group)
            c.amount += distributed;
    }

    public void connectTo(Container other) {
        if (group == other.group) return;

        int size1 = group.size(),
            size2 = other.group.size();
        double tot1 = amount * size1,
               tot2 = other.amount * size2,
               newAmount = (tot1 + tot2) / (size1 + size2);
        group.addAll(other.group);

        for (Container c: other.group)
            c.group = group;
        for (Container c: group)
            c.amount = newAmount;
    }
}
```

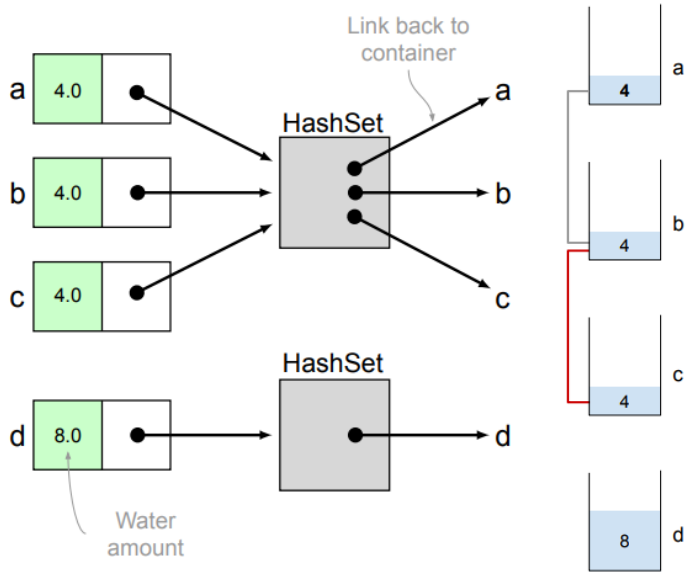
amount è la quantità contenuta in questo container, mentre *group* sono i container connessi direttamente o indirettamente a questo.
Nota che questo *Set* non sarà mai vuoto poiché conterrà almeno *this*

Se è già connesso non fa nulla

Calcola i nuovi campi ed aggiunge gli elementi dell'altro gruppo a questo

Aggiorna i gruppi degli altri container e la quantità d'acqua di ogni container

Forniamo di seguito un esempio del memory layout del caso d'uso numero 2 con questa implementazione:



Efficienza di tempo

Nell'ottica di efficienza di tempo per un codice le linee guida sono:

0) **Hai davvero bisogno di più spazio**? Nella realtà si preferisce sempre la leggibilità del codice (e quindi manutenibilità) che l'efficienza del codice in se (ovviamente ci possono essere dei casi particolari per cui serva fare questa ottimizzazione)
1) **Complessità asintotica**: valutare la velocità all'aumento della dimensione degli ingressi.
2) **Profiling** (profilatura) **e ottimizzazione**: fare una stima dei seguenti aspetti: **usage profile** (quante volte il cliente chiama ogni metodo), e **runtime profile** (Quale metodo impiega più tempo?). Dopo queste stime si va ad ottimizzare il metodo (o i metodi) più comune/costoso.

Detto ciò, la nostra implementazione Reference ha la seguente complessità:

$getAmount$	$\mathrm{O}(1)$
$connectTo$	$\mathrm{O}(n)$
$addWater$	$\mathrm{O}(n)$

Seguendo l'approccio precedentemente descritto possiamo fare delle ottimizzazioni. Più precisamente abbiamo due possibili alternative:

Reference:

Method	Time complexity
getAmount	$O(1)$
connectTo	$O(n)$
addWater	$O(n)$

??

incomparable

Alternative 2:

Method	Time complexity
getAmount	$O(n)$
connectTo	$O(1)$
addWater	$O(1)$

Dominates! *(better)*

Alternative 1:

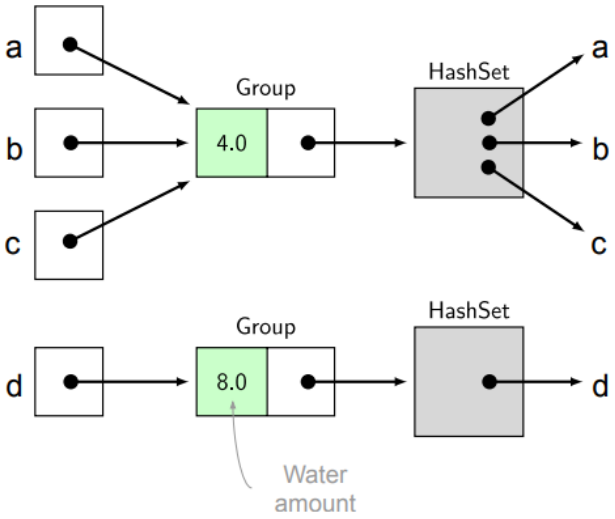
Method	Time complexity
getAmount	$O(1)$
connectTo	$O(n)$
addWater	$O(1)$

L'alternativa 2 potrebbe sembrare migliore, ma per deciderlo bisogna avere un Usage Profile di quel metodo; infatti, se il metodo più usato dal cliente fosse *getAmount* allora sarebbe migliore l'alternativa 1.

Speed1

Per aggiungere acqua in tempo costante basta separare il gruppo di oggetti, più precisamente avere in *Container* un singolo campo con una classe innestata:

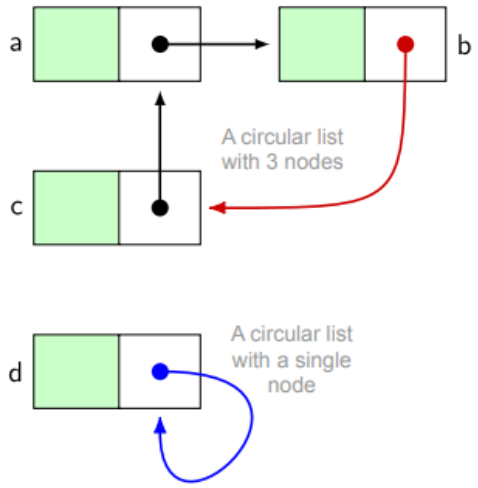
```
// A single field:
Group group = new Group(this);
// A nested class:
private static class Group {
    double amountPerContainer;
    Set<Container> members;
    Group(Container c) {
        members = new HashSet<>();
        members.add(c);
    }
}
```

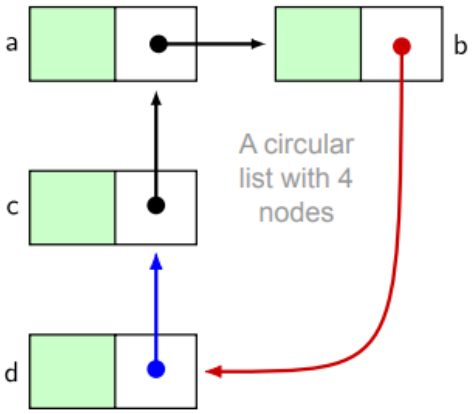


Method	Time complexity
getAmount	$O(1)$
connectTo	$O(n)$
addWater	~~$O(n)$~~ ⇦ $O(1)$

Speed2

L'altra ottimizzazione è quella di avere in tempo costante sia *addWater* che *connectTo* (a discapito di *getAmount* che diventa lineare). Questo è possibile grazie alle liste circolare, che possono essere unite in tempo costante, ed ad una pratica chiamata **laziness**; ovvero, ritardare il più possibile la combinazione. Nel nostro caso sembra che ciò violi le specifiche (*connectTo* non suddivide l'acqua), ma in realtà è lecito; infatti, il cliente non può accorgersi di ciò fino a che non richiama *getAmount* il quale è incaricato di equiparare i container prima di restituire il risultato.






```
// I campi:
double amount;
Container next;

// Connessione di due container
public void connectTo(Container other) {
    Container oldNext = next;
    next = other.next;
    other.next = oldNext;
}
```

Method	Time complexity
getAmount	$O(n)$
connectTo	$O(1)$
addWater	$O(1)$

Attenzione: questa implementazione, oltre a non modificare il parametro *amount*, **non controlla che *this* e *other* non siano già collegati** (non è possibile farlo in tempo costante).

Speed3 (union-find tree)

Ovviamente non possiamo avere tutti i metodi in tempo costante (questo fatto è dimostrabile), ma possiamo avvicinarci di molto, grazie all'ausilio dei cosiddetti alberi union-find, usati per rappresentare insiemi disgiunti.

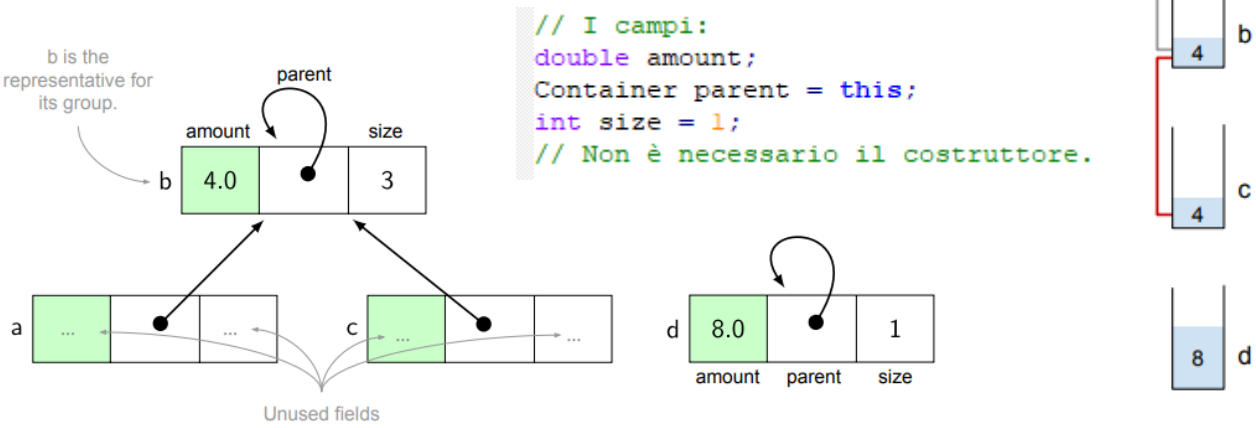
Dato un insieme di *n* elementi inizialmente isolati, gli alberi union-find permettono due operazioni:

- **Union**: dati due elementi ne unisce gli insiemi
- **Find**: Dato un elemento, restituisce il rappresentante di quell'insieme

Risulta evidente che due insiemi saranno uguali se avranno lo stesso rappresentante.

L'implementazione concreta di questi alberi è la seguente:

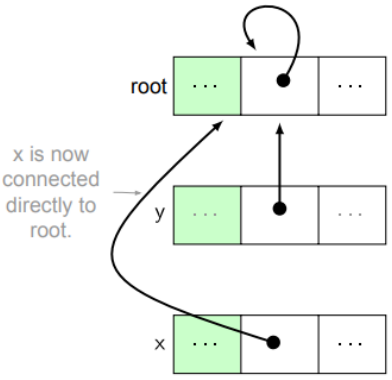
- Ogni insieme è un parent-point tree; ovvero, un albero dove ogni nodo ha solo il puntatore verso il padre (la radice avrà come genitore sé stessa)
- Nella nostra implementazione ogni nodo avrà tre campi: *amount*, *parent* e *size*. Le informazioni andranno lette solo alla radice (e quindi basta aggiornale lì)



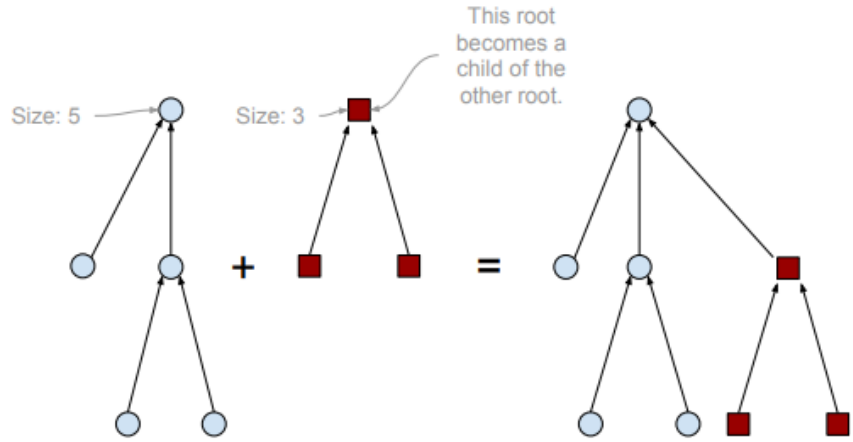
Gruppi di container come alberi union-find:

- ***getAmount*** (operazione find): si risale alla radice dell'albero e si restituisce il campo *amount* della radice. Il tutto mentre si applica il **path compression**; ovvero, mentre si naviga da nodo a nodo, si trasforma ogni nodo in un figlio diretto della radice (praticamente si schiaccia l'albero):

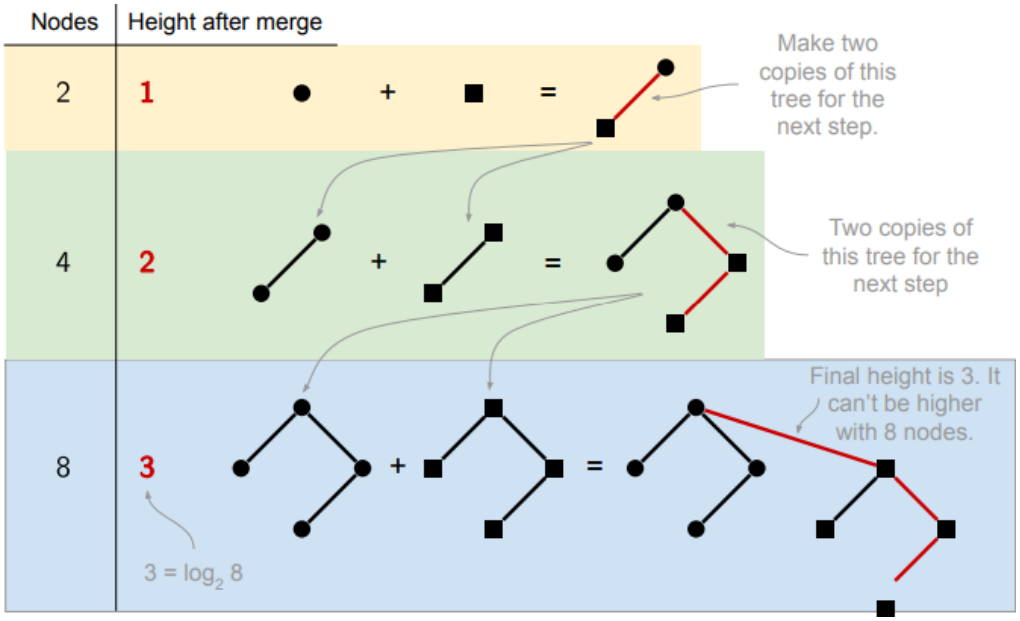
```
public double getAmount() {
    Container root = findRootAndCompress();
    return root.amount;
}

private Container findRootAndCompress() {
    if (parent != this)
        parent = parent.findRootAndCompress();
    return parent;
}
```



- ***connectTo*** (operazione union): Si risale alla radice di entrambi gli alberi (e si controlla che questi siano differenti) e si uniscono i due alberi trasformando la radice del più piccolo nel figlio dell'altra radice; quindi, applicando la **link-by-size policy**:



Questa politica assicura che l'altezza sia, nel caso peggiore, al più logaritmica:



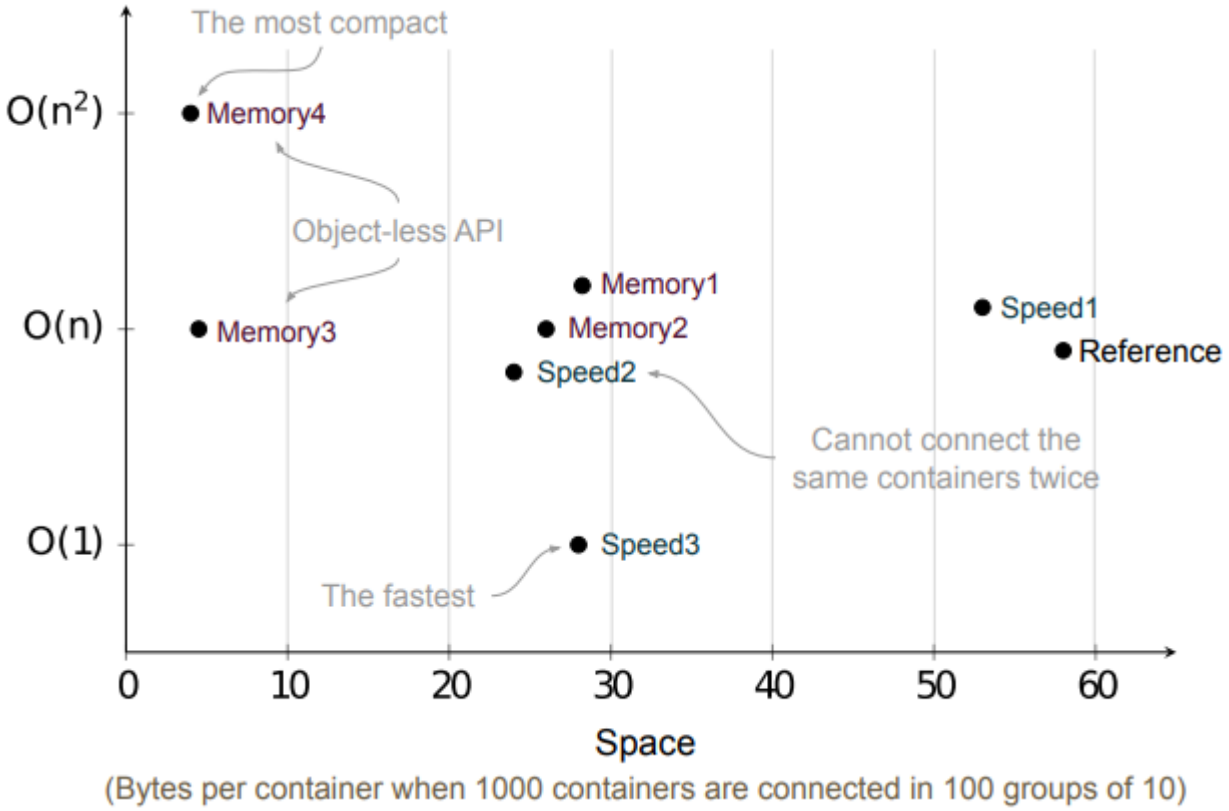
Si potrebbe pensare che nel caso peggiore questa implementazione abbia complessità $O(\log n)$ in tutti i suoi metodi, ma in realtà per algoritmi che fanno investimenti per futuri benefici (come nel nostro caso) la complessità è ammortizzata (l'abbiamo studiata negli ArrayList), rendendo anche nel caso peggiore la nostra complessità praticamente costante.

Grazie alla politica link-by-size e al path compression, la complessità di qualsiasi sequenza *m* delle operazioni union e find è **al più lineare in *m***:

1) Teorema di Tarjan: qualsiasi sequenza di *m* operazioni union e find prende al più tempo $O(m \cdot \alpha(n))$, dove $\alpha(n)$ è l'inverso della funzione di Ackermann
   - $\alpha(n)$ è al più 4 per un input *n* di 10^{80}
   - Il costo di una singola operazione è essenzialmente costante

Comparazione delle implementazioni

Di seguito mostriamo un grafico dove compariamo le varie implementazioni (comprese quelle future dove andremo ad ottimizzare la memoria) così da avere una visione sommaria dei pro e contro delle varie implementazioni. Ovviamente per ogni ottimizzazione andremo a rappresentare la complessità del metodo più lento:



Forniamo di seguito anche degli esperimenti, riportando il tempo dell'esecuzione dei programmi nelle varie implementazioni svolte fino ad ora:

Esperimento 1:

1) Creazione di 20k container e aggiunta di acqua
2) Connessione dei container in due coppie da 10k
3) Aggiunta di acqua in ogni insieme
4) Query di *amount* per ogni insieme
5) Connessione dei due insiemi, aggiunta di acqua e query dell'*amount*

Version	Time (msec)
Reference	2 300
Speed1	26
Speed2	505
Speed3	6

Esperimento 1:

1) Creazione di 20k container e aggiunta di acqua
2) Connessione dei container in due coppie da 10k
3) Aggiunta di acqua in ogni insieme
4) ~~Query di *amount* per ogni insieme~~
5) Connessione dei due insiemi, aggiunta di acqua ~~e query dell'*amount*~~
6) Query dell'*amount* (qui chiamiamo 1 sola volta *getAmount,* nell'esperimento 1 invece 20k volte)

Version	Time (msec)
Reference	2 300
Speed1	25
Speed2	4
Speed3	5

Efficienza di spazio

È utile ottimizzare lo spazio di memoria poiché potremmo avere poco spazio oppure molti dati (o entrambe!). Ad esempio, se stiamo lavorando su sistemi embedded (poco spazio) oppure dobbiamo gestire bilioni di container!

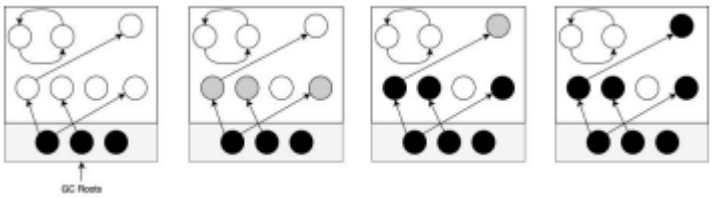
Prima di passare alle varie ottimizzazioni della nostra Reference vediamo la memoria occupata dagli oggetti in Java. Ogni oggetto in java include un **object header** con delle informazioni ausiliare. L'header supporta le seguenti funzionalità: consapevolezza del tipo e riflessione, multithreading, garbage collection.

Lo **spazio di un riferimento** in teoria è lo spazio dell'indirizzo di memoria (quindi 64bit per architetture da 64 bit, e 32 per architetture a 32 bit). Ma nella pratica anche per le architettura a 64bit i riferimenti occupano sempre 32 bit; dunque, si utilizza la COOPs (compressed ordinary object pointers); ovvero, quando si utilizza un indirizzo, si aggiunge 3 zeri alla fine così da avere 35 bit per indirizzo (questo è possibile poiché molti programmi non utilizzano più di 32GB di memoria). Questa ottimizzazione è attiva di default in HotSpot ma la si può disattivare con una linea di comando nella JVM.
Inoltre, poiché ogni oggetto deve conosce il proprio tipo dinamico (usato in *istanceof*, *getClass* o altre operazioni riflessive) l'header contiene anche **un puntatore all'oggetto di tipo Class**.

Per quanto riguarda l'overhead del multi-threading, in teoria Java assegna un monitor ad ogni oggetto (simile al mutex) con cui si accede tramite la chiave *synchronized*. Nella pratica la versione corrente di HotSpot instanzia il monitor on demand riducendo (ma non eliminando) l'overhead di memoria.

L'overhead del garbage collection, invece, in teoria dovrebbe essere un contatore per ogni oggetto (e quindi un altro riferimento) ma l'HotSpot implementa una politica di tracing (mark-and-sweep) e generazionale. Più precisamente:

- Tracing: invece del contatore si seguono tutti i riferimenti partendo dalle radici del garbage collector (praticamente si scopre un grafo)



- Generazionale: Sfrutta il fatto che in un programma un oggetto o ha vita breve, oppure molta lunga. Quindi gli oggetti sono arrangiati in gruppi chiamati generazioni, basati sul tempo passato dalla loro creazione e il garbage collector semplicemente controlla meno spesso quelli che hanno una aspettativa di vita più lunga (tanto non è un problema che un oggetto rimanga un po' in più).

Altri overhead sono quelli di allineamento e padding. Nella maggior parte delle architetture l'accesso in memoria è più efficiente se essa è word-aligned (praticamente gli indirizzi sono multipli di 8).
L'allineamento sfrutta una tecnica simile alla COOPs per tenere gli indirizzi word-aligned, aggiungendo spazi vuoti dove necessario (padding).

Per semplicità ignoreremo l'alignement e il padding, dunque riassumendo il nostro overhead avrà un totale di **12 byte** ad oggetto (4 byte per il puntatore all'oggetto Class e 8 per Monitor + Garbage Collection).

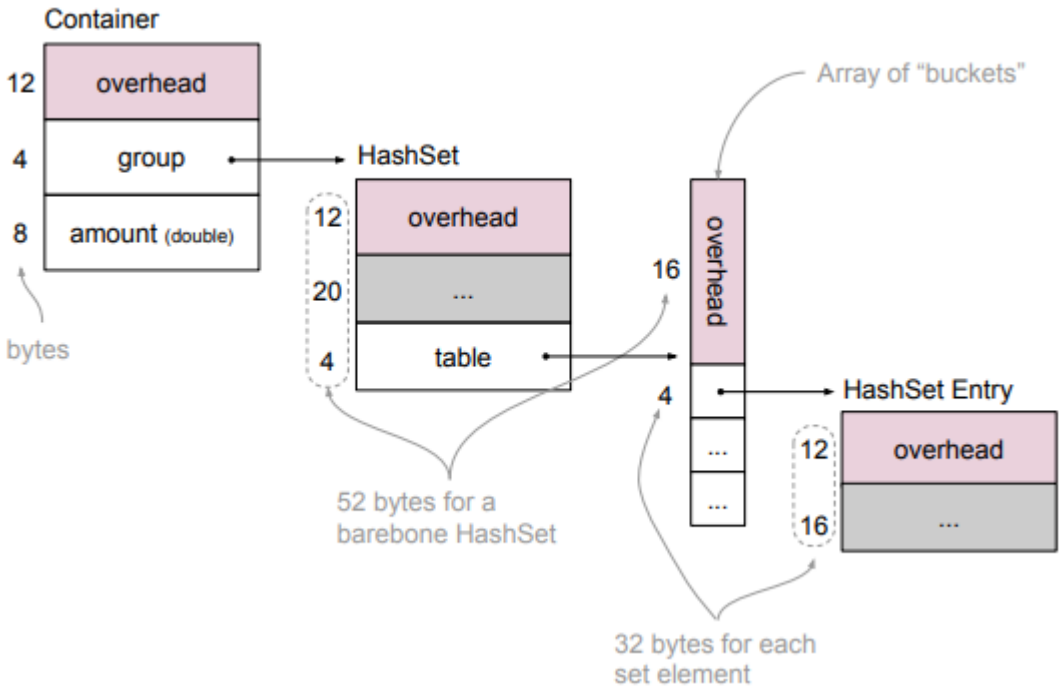
Per quanto riguarda invece gli array, essi avranno un overhead di **16 byte** ciascuno, poiché ai 12 byte precedenti vanno aggiunti 4 byte addizionali per la size. Il tipo statico delle loro celle non ha overhead addizionale essendo questa informazione parte del loro tipo.

Facciamo un confronto con il linguaggio C: quest'ultimo non supportando il multithreading e non essendoci il garbage collector ha un overhead di default pari a 0 per gli oggetti. Infatti, la riflessione (conosciuta come RTTI in C) è solo per classi con almeno un metodo virtuale.

Memory layout di Reference

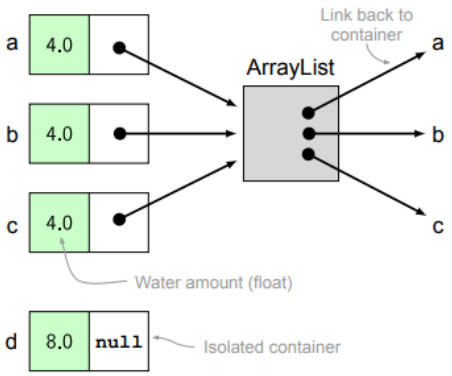
Passiamo ora a calcolare lo spazio della nostra implementazione di riferimento. Ma prima di riportare il memory layout dettagliato di Reference si riporta in tabella i requisiti di memoria in due scenari convenzionali:

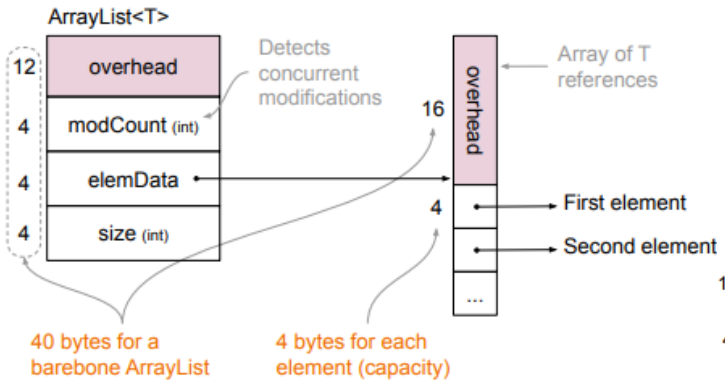
Scenario	Calcolo della Size	Size (byte)
1000 container isolati	$1000 * (12 + 8 + 4 + 52 + 32)$	108000
100 gruppi da 10	$1000 * (12 + 8 + 4) + 100 * (52 + 10 * 32)$	61200

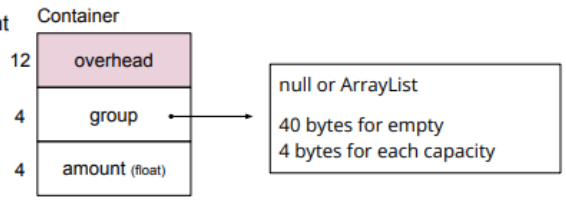


Memory1

Una ottimizzazione immediata potrebbe essere quella di passare da double a float per il campo *amount* e da un HashSet ad un ArrayList per il campo *Group* (questo metodo mantiene la stessa complessità di Reference), così da avere i seguenti miglioramenti:







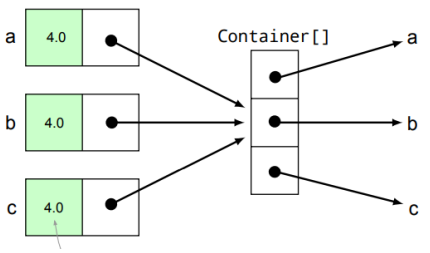
La memoria richiesta di Memory1 sarà dunque:

Scenario	Calcolo della Size	Size (byte)	% di Reference
1000 isolati	$1000 * (12 + 4 + 4)$	20000	19%
100 gruppi da 10	$1000 * (12 + 4 + 4) + 100 * (40 + 10 * 1.25 * 4)$	29000	47%

Nota: il moltiplicativo 1.25 è la capacità in eccesso prevista da ArrayList (le celle ancora non usate).

Memory2

Questa implementazione consiste nel rappresentare un gruppo come un array di container, invece di un ArrayList. Questa modifica implica la perdita di tutti i metodi offerti da ArrayList con la ovvia conseguenza di dover implementare i metodi che ci servono, come ad esempio il ridimensionamento dell'array.



Nella tabella seguente riportiamo la memoria richiesta dalle comuni collezioni, assumendo che la capacità di ArrayList e HashSet sia uguale alla loro size. La seconda colonna è meno rilevante della terza poiché la prima citata è la dimensione della collezione che è ovviamente in un numero ridotto rispetto al numero delle celle:

Tipo	Size (scheletro)	Size (di ogni elemento extra)
array	16	4
ArrayList	40	4
LinkedList	24	24
HashSet	52	32

Ne consegue che usare un array invece di un ArrayList si ha un risparmio di 34 byte

La memoria richiesta di Memory2 sarà dunque:

Scenario	Calcolo della Size	Size (byte)	% di Reference
1000 isolati	$1000 * (12 + 4 + 4)$	20000	19%
100 gruppi da 10	$1000 * (12 + 4 + 4) + 100 * (16 + 10 * 4)$	25600	42%

È evidente che non vale la pena usare questo tipo di rappresentazione poiché il risparmio di memoria è poco e quindi non vale lo sbatti di dover gestire l'array

Memory3

Nelle rappresentazioni seguenti vedremo come risparmiare molto spazio in memoria cambiato completamente rappresentazioni. Ovvero, usare un **object-less API**.

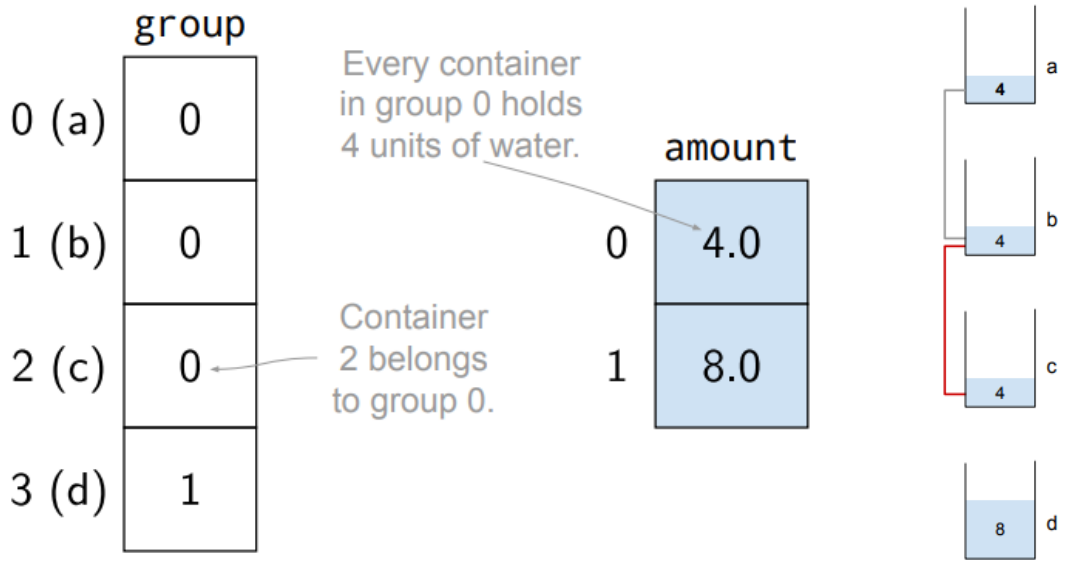
Rappresenteremo i nostri container, invece che con oggetti, tramite interi (che saranno gli ID dei container). Ne consegue che i costruttori e i metodi diverranno 4 metodi statici non orientati ad oggetti:

```
// Caso d'uso di questa implementazione:
int a = Container.newContainer(),
    b = Container.newContainer(),
    c = Container.newContainer(),
    d = Container.newContainer();

Container.addWater(a, 12);
Container.addWater(d, 8);
Container.connect(a, b);
System.out.println(Container.getAmount(a));
```

Ogni container è rappresentato da un suo ID (*containerID*) ed ogni gruppo è rappresentato da un suo ID (*groupID*), il vettore amount verrà inizializzato come un array vuoto per non creare overhead:

```
public class Container {
    // Da containerID a groupID
    private static int group[] = new int[0];
    // Da groupID all'amount contenuto in ogni container
    private static float amount[] = new float[0];

    public static float getAmount(int containerID) {
        int groupID = group[containerID];
        return amount[groupID];
    }
    ...
}
```

Il memory layout sarà rappresentato dunque da due array:



- Un array group dove gli indici rappresenteranno l'ID del container e il contenuto l'ID del gruppo al quale appartiene il container
- Un array amount dove l'indice rappresenta l'ID del grouppo e il contenuto la quantità d'acqua presente in **ogni** container di quel gruppo.

Precedentemente abbiamo dato un'implementazione di *getAmount* (unico metodo costante). Poiché gli altri metodi sono abbastanza complessi ne descriveremo solo teoricamente l'implementazione:

- Metodo ***newContainer:***
  - Aggiunge un nuovo *containerID* e un nuovo *groupID*
  - Estende entrambi gli array
  - Tempo **lineare sul numero totale di container** (e non sul numero di container nel gruppo)
- Metodo ***addWater***:
  - Conta la size del gruppo (scorrendo l'array)
  - Aggiorna l'amount di ogni container del gruppo nell'array
  - Tempo lineare sul numero totale di container (nel caso peggiore potremmo avere in un gruppo un container all'inizio ed uno alla fine dell'array, infatti non è detto che i container in un gruppo siano in sequenza)
- Metodo ***connect***:
  - Unisce due gruppi
  - Rimuove il *groupID* di un gruppo
  - Per scoprire tutti i container di un gruppo, bisogna iterare su **tutti i container**
  - Tempo lineare sul numero totale di container

Memoria richiesta di Memory3:

Scenario	Calcolo della Size	Size (byte)	% di Reference
1000 isolati	$4 + 16 + 1000 * 4 + 4 + 16 + 1000 * 4$	8040	7%
100 gruppi da 10	$4 + 16 + 1000 * 4 + 4 + 16 + 100 * 4$	4440	7%

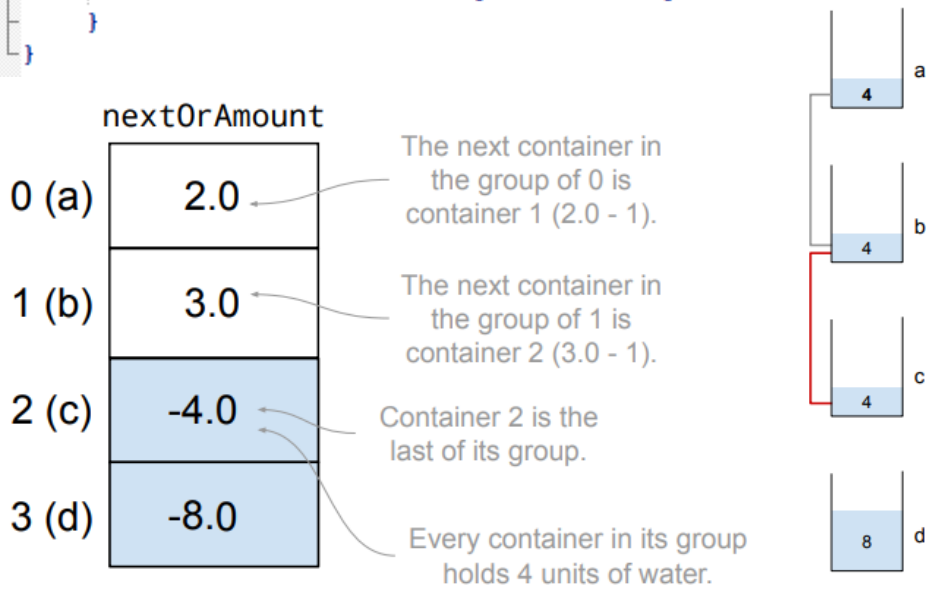
Quindi anche se quasi tutti i metodi sono lineari risparmiamo ben il 93% di spazio rispetto Reference

Memory4

Possiamo risparmiare ancora più spazio se usassimo un singolo array di float che rappresenta sia i gruppi che la quantità di acqua in ogni container. Ciò è possibile poiché possiamo sfruttare il fatto che la quantità d'acqua nei container non è mai negativa e dunque abbiamo i float < 0 liberi e usabili.

Più precisamente, quando il valore è positivo esso rappresenta l'indice del **prossimo container in questo gruppo**, con un bias (tipo un offset) di +1 (questo ci consente di utilizzare anche la cella di indice 0, poiché zero è un valore valido per amount). Quando, invece, il valore è negativo o zero rappresenta l'opposto della quantità d'acqua presente in **ogni** contenitore di quel gruppo.

```
public class Container {
    private static float nextOrAmount[] = new float[0];

    public static float getAmount(int containerID) {
        while (nextOrAmount[containerID] > 0)
            containerID = (int)nextOrAmount[containerID] - 1;
        return -nextOrAmount[containerID];
    }
}
```



Questo tipo di implementazione (oltre al mal di testa per implementare *connectTo*) ha delle debolezze. In primis possiamo rappresentare solo una quantità limitata di container; infatti, l'intervallo di interi rappresentabile con i float va da 0 a 2^{24} (uninterrupted integer range), dopo tale cifra non tutti gli interi sono rappresentabili (si ricorda l'esempio con un miliardo e un miliardo e uno al capitolo 1).

Un'altra debolezza sta nel tempo di esecuzione; infatti, questo tipo di rappresentazione ci costringe a dover visitare a ritroso l'array.
Vediamo in maniera teorica l'implementazione degli altri due metodi:

- Metodo ***addWater***:
  - Trovare l'ultimo container del gruppo
  - Contare la size del gruppo
  - Tempo quadratico sul numero totale di container
- Metodo ***connectTo***:
  - Trovare tutti i container in **entrambi** i gruppi
  - Tempo quadratico sul numero totale di container

Method	Time complexity
getAmount	$O(n)$
connectTo	$O(n^2)$
addWater	$O(n^2)$

In conclusione, rappresentiamo la memoria richiesta da tutte le implementazioni di questo capitolo e Reference. Ricordiamo che Memory3 e Memory4 sono object-less API.

Scenario	Versione	Byte	% di Reference
1000 isolati	Reference	108000	100%
	Memory1	20000	19%
	Memory2	20000	19%
	Memory3	8040	7%
	Memory4	4020	4%

100 gruppi da 10	Reference	61200	100%
	Memory1	29000	47%
	Memory2	25600	42%
	Memory3	4440	7%
	Memory4	4020	7%

Si rimanda anche alla visione del grafico nel paragrafo "comparazione delle implementazioni".

17. Lambda espressioni

Programmazione funzionale

Il concetto di lambda espressione deriva dal lambda calcolo, il quale è il primo linguaggio di programmazione basato su funzioni nel senso matematico; ovvero, dato lo stesso input si ha lo stesso output (stateless, quindi senza variabili, assegnamenti o loop). I principi del lambda calcolo sono quindi, quelli di un linguaggio funzionale (come LISP e ML) e dunque basati su paradigma funzionale:

- Il programma e le sue componenti sono funzioni o meglio espressioni e quindi l'esecuzione diventa una valutazione di funzioni. Nella versione funzionale pura non ci sono variabili né assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione.
- Invece della comunicazione c'è la composizione, dunque posso chiamare una funzione di questo tipo: $f\big(g(a), h(b), g(c)\big)$, con la conseguenza che, se *g* e *h* sono funzioni stateless possono essere valutate anche in parallelo (uno dei vantaggi del paradigma funzionale).

Interfacce funzionali in java

Nelle interfacce in java c'è sempre stata la possibilità di inserire campi *public static final*, da java 8 è anche possibile avere metodi *public static* con la conseguenza che alcune classi come *Math* possono ora essere utilizzate come interfacce. Un'altra importante aggiornamento di Java sulle interfacce è stata introdurre la possibilità di creare metodi concreti che possono essere sovrascritti (tramite la keyword *default*).

Dunque, ora java supporta l'ereditarietà multipla:

```
interface A {
    default void foo() {
        System.out.println("I have an implementation!");
    }
}
interface B {
    default void foo() {
        System.out.println("I have another implementation!");
    }
}
class X implements A, B { } // Ambiguo: errore in compilazione
                            // X deve avere l'override di foo
```

Quindi ora, grazie a questi metodi *default*, le interfacce possono oltre ad offrire firme (*signature*) anche offrire comportamenti (codice). Ne segue che adesso l'unica differenza dalle classi astratte è che le interfacce sono ancora stateless (non possono avere campi instanza e sono prive di attributi). Riassumendo, le interfacce forniscono firme, comportamenti ma non stato; le classi astratte forniscono comportamenti e stato.

Questa introduzione dei metodi *default* ha permesso a java di inserire nuovi metodi senza interferire sulle classi che le implementano (retrocompatibilità). Ne è un esempio la classe *Comparator* che in Java 7 forniva solo un metodo astratto (*compare*) ma da Java 8 sono stati aggiunti 9 metodi statici e 7 di default

- Introduciamo un metodo di default in *Comparator<T>*:
  ```
  default Comparator<T> thenComparing(Comparator<? super T> other)
  ```
  Questo restituisce un nuovo comparatore composto da questo comparatore (*this*) con un altro

comparatore (*other*) in ordine lessicografico; ovvero, dati due oggetti da comparare, prima si valuta con il comparatore *this* e se questo restituisce 0, vengono valutati dal comparatore *other*.

Il seguente esempio riassume tutto ciò che è possibile fare con le interfacce:

```
public interface Scalable {

    // implicitamente public abstract
    void setScale(double scale);

    // Implicitamente public static final
    double DEFAULT_SCALE = 1.0;

    // Ora possibile (da Java 8):

    // Implicitamente public
    static boolean isScalable(Object obj) {
        return obj instanceof Scalable;
    }

    // Implicitamente public
    default void resetScale() {
        setScale(DEFAULT_SCALE);
    }
}
```

Interfacce funzionali pure

Qualunque interfaccia con un singolo metodo astratto viene definita **interfaccia funzionale pura** (sono permessi metodi statici e di default); più precisamente, un interfaccia funzionale è definita pura se è intesa e progettata per essere implementata da classi prive di stato (quindi è una definizione a livello di contratto). Un esempio è *Comparator* che è funzionale pura, mentre non lo sono *Comparable* o *Runnable*.

Le interfacce funzionali pure di java rispettano i paradigmi della programmazione funzionale, queste interfacce sono così importanti che Java offre la possibilità di annotarle con *@FuncionalInterface*. Una interfaccia con tale annotazione viene considerata funzionale pura e il compilatore controlla che rispetti la proprietà di avere un singolo metodo astratto (ovviamente non può controllare che rispetti il paradigma funzionale).

In Java sono presenti più di 40 interfacce funzionali pure, poste nel pacchetto *java.util.function*, ne sono da esempio le seguenti interfacce:

```
// Una funzione che accetta un oggetto
@FunctionalInterface
pubic interface Consumer<T> {
    void accept (T t);
}
```

```
// Una funzione che produce un oggetto
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Lambda espressioni

La lambda espressione è una sintassi compatta per implementare le interfacce funzionali pure, ed è un ottima alternativa alle classi anonime. Sia ad esempio il seguente codice (usiamo una classe anonima):

```
Comparator<Employee> byName = new Comparator<Employee>() {
    public int compare(Employee a, Employee b) {
        return a.getName().compareTo(b.getName());
    }
};
```

Tramite l'uso della lambda espressione, diventa:

```
Comparator<Employee> byName = (Employee a, Employee b) -> {
    return a.getName().compareTo(b.getName());
};
```

Ne consegue che la sintassi di una lambda espressione è la seguente:

- **Parametri**: possono essere una lista di parametri, ad esempio *(int a, int b)*, la quale può semplicemente essere abbreviata con *(a, b)*; oppure un singolo parametro *(a)* o *a* (senza parentesi), o ancora nessun parametro *()*
- **Corpo**: può essere un blocco di codice, in questo caso va inserito tra graffe, oppure una semplice espressione (non è necessario mettere le parentesi)

Riprendendo il nostro esempio precedente, esso può essere ulteriormente ridotto con quanto detto sopra:

```
Comparator<Employee> byName =
    (a, b) -> a.getName().compareTo(b.getName());
```

Le lambda espressioni sono possibili solo in determinati contesti dove la type inference può estrapolare abbastanza informazioni. Ovvero, è possibile utilizzarle **solo** per interfacce funzionali pure dove la type inference è in grado di identificare il metodo implementato (nell'esempio precedente è l'override di *CompareTo*) e i parametri di tipo (se omessi).

I contesti che hanno abbastanza informazioni per identificare l'interfaccia funzionale ricevuta sono:

- A destra di un assegnamento: *Consumer<String> c =* ***lambda***
- Parametro attuale di un metodo o un costruttore: *new Thread(**lambda**)*
- Argomento di un "*return*": *return **lambda***
- Argomento di un cast: *(Consumer<String>) **lambda***

È convenzione usare le lambda espressioni solo per codici brevi; utilizzarlo per blocchi di più di 3/4 righe di codice diventerebbe poco leggibile ed è quindi sconsigliato.

Cattura delle variabili

La lambda espressione può avere accesso a **campi statici** di qualsiasi classe (ovviamente), a **variabili locali** di classi in cui è annidata e a **campi istanza** di oggetti che la contengono (negli ultimi due casi cattura).

Più precisamente, nel caso di **variabili locali** in cui la lambda espressione è annidata, quest'ultima ne ha l'accesso purché siano *effectively final* (stessa regola delle classi anonime). Praticamente la lambda espressione memorizza **una copia della variabile** e la "cattura". Ogni valutazione a runtime può o non può generare un nuovo oggetto, se la variabile usata nella lambda espressione è dichiarata final allora non viene istanziato un nuovo oggetto ad ogni chiamata altrimenti si (ci sono casi particolari come variabili usate come se fossero final, ma non definiremo questo confine).

Per quanto riguarda i campi, la lambda espressione può accedere all'oggetto includente (***this***) e ai suoi **campi istanza**, in questi casi cattura sempre la variabile.

```
class Test {
    public Consumer<String> foo() {
        return (msg -> System.out.println(msg + this));
    }
}
```

In questo caso la lambda espressione cattura l'oggetto *Test* corrente (si noti la differenza con le classi anonime, in cui *this* si riferisce alla classe anonima e quindi avremmo dovuto scrivere *Test.this* per riferirci all'oggetto). Per quanto riguarda i campi istanza, la lambda espressione memorizza il **riferimento dell'oggetto** in cui è annidata e "cattura" l'istanza corrente (*Instance-capturing lambda expression*), simile alla cattura della variabile *this*. Ogni valutazione a runtime genera un nuovo oggetto.

Esempi sono disponibili in questo link: bitbucket.org/mfaella/…/LambdaImplementation.java

Riassumendo, se non cattura una variabile allora la lambda espressione **non** crea un nuovo oggetto ogni volta che viene invocata, rendendo quindi questa sintassi più efficiente di una classe anonima (che istanzia un nuovo oggetto a prescindere dalla situazione).

Method Reference

Una **method reference** è una espressione che denota un metodo (rappresenta il riferimento di quel metodo); simile ai puntatori a funzioni in C/C++. Questa sintassi risulta essere più efficiente della riflessione.

Presentiamo di seguito la sintassi per riferirsi ad un metodo:

- Metodo statico: *Employee::getMaxSalary*
- Metodo istanza, istanza non specifica: *Employee::getSalary*
- Metodo istanza, istanza specifica *mike::getSalary*(dove *mike* è un *Employee*)
- Costruttore: *Employee::new*

Sintassi meno frequente (di "nicchia"):

- Metodo istanza della superclasse: *super::foo*
- Costruttore di Array: *A[]::new*

Une method reference è una espressione che non ha un tipo intrinseco ma **il tipo viene dedotto dal contesto**; in maniera analoga alle lambda espressioni.

I contesti di una method reference sono esattamente quelli di una lambda espressione (il tipo ricevente *T* deve essere una interfaccia funzionale con unico metodo astratto compatibile con il method reference):

contesto	**esempio**	**Tipo ricevente**
A destra di un assegnamento	*T var =* ***<method ref>***	*T*
Parametro attuale di un metodo o un costruttore	*foo(****<method ref>****)*	il tipo corrisponde al parametro formale
Argomento di un "*return*"	*return* ***<method ref>***	il tipo di ritorno del metodo corrente
Argomento di un cast	*(T)* ***<method ref>***	*T*

Sia le lambda espressioni che le method reference sono strumenti pratici per passare codice (scrivere funzioni di ordine superiore).
Da classe a method reference:

- Classe "normale":

```
class Printer implements Consumer<String> {
    @Override
    public int accept(String s) {
        System.out.println(s);
    }
}
Consumer<String> printer = new Printer();
```

- Classe anonima:

```
Consumer<String> printer = new Consumer<String>() {
    @Override
    public int accept(String s) {
        System.out.println(s);
    }
};
```

- Lambda espressione:
```
Consumer<String> printer = s -> System.out.println(s);
```
- **Method reference**
```
Consumer<String> printer = System.out::println;
```

Al seguente link potete trovare vari esempi: bitbucket.org/mfaella/.../MethodReferences.java

18. Introduzione al multithreading

I thread

I **thread**, o processi leggeri, sono flussi di esecuzione all'interno di un processo in corso; in altre parole, un processo può essere suddiviso in vari thread, ciascuno dei quali rappresenta un flusso di esecuzione indipendente dagli altri. I thread appartenenti allo stesso processo condividono quasi tutte le risorse, come la **memoria** e i file aperti. Mentre, non condividono quelle risorse che consentono ad un thread di avere un flusso di esecuzione indipendente; ovvero, il program counter e lo stack.

Java è stato il primo tra i linguaggi di programmazione maggiormente utilizzati ad offrire un **supporto nativo** ai thread, a differenza dei linguaggi C/C++ che per supportare i thread necessitano di librerie esterne, spesso fornite dal sistema operativo (come la libreria per i thread POSIX).
Siccome la virtual machine di Java funge anche da sistema operativo per i programmi Java, essa offre in maniera nativa il supporto ai thread.

La classe Thread

Esamineremo due modi alternativi di creare un thread in java, entrambi i modi sono supportati dalla **classe *Thread***. Nota che per evitare confusione tra i thread e gli oggetti della classe *Thread*, chiameremo "thread di esecuzione" i primi e "oggetti thread" i secondi.*

In Java, ad ogni thread di esecuzione è associato un oggetto thread ma il viceversa non è sempre vero, in quanto un oggetto thread può non avere un corrispondente thread di esecuzione, perché quest'ultimo non è ancora partito, oppure perché è già terminato (questo succede perché la vita dell'oggetto thread è più lunga di quella del thread di esecuzione).

Una applicazione Java termina quando **tutti** i suoi thread sono terminati. Ogni applicazione Java parte con almeno un thread, detto thread principale (*main thread*), che esegue il metodo main della classe di partenza. Anche al thread principale è associato, in maniera automatica, un oggetto thread; in seguito, vedremo come ottenere un riferimento a questo oggetto.

Metodi utili della classe Thread

```
public static void sleep(long millis) throws InterruptedException
```

- Tale metodo statico mette in attesa il thread corrente (cioè, quello che invoca *sleep*) per un dato numero di millisecondi
- Se l'attesa viene interrotta il metodo lancia l'**eccezione verificata** *InterruptedException.*
  - Questa è una caratteristica comune a tutti i metodi cosiddetti "bloccanti", che cioè possono mettere in attesa un thread
  - Quando si cattura l'eccezione *InterruptedException*, è buona norma terminare il thread di esecuzione corrente (si veda il paragrafo "interrompere un thread")
- Se ci si trova nel metodo *run* di un oggetto thread, per terminare il thread corrente è sufficiente utilizzare *return*

```
public static Thread currentThread()
```

- Restituisce l'oggetto thread corrispondente al thread di esecuzione che l'ha invocato
- Con questo metodo è possibile anche ottenere un riferimento all'oggetto thread corrispondente al thread principale (quello che esegue inizialmente il metodo main)

```
public final void join() throws InterruptedException
```

- Il metodo *join* interagisce con due thread (sia oggetti, sia thread di esecuzione):
  - thread 1: il thread che lo chiama, cioè il flusso di esecuzione che invoca *join*
  - thread 2: il thread sul quale è chiamato, cioè il thread corrispondente all'oggetto puntato da *this*
- Praticamente, se in un thread 1 (il thread corrente, quello che esegue il *join*) chiamo *t2.join()*, dove t2 è la variabile di un thread 2, il metodo *join* **mette in attesa il thread 1 fino alla terminazione del thread 2**
  - Se il thread 2 è già terminato, il metodo join ritorna immediatamente
  - Se il thread 2 non è ancora partito, il metodo join aspetta la partenza e poi la terminazione
- Questo metodo è un metodo bloccante e quindi lancia l'eccezione verificata *InterruptedException* se l'attesa viene interrotta
- Il metodo *join* svolge un compito analogo alla system call waitpid dei sistemi Unix, nonché della funzione pthread_join dello standard POSIX thread

Creazione di un thread

Il primo modo di creare un thread di esecuzione consiste nei seguenti passi:

1) Creare una classe *X* che estenda *Thread*
2) Affinché il nuovo thread di esecuzione faccia qualcosa, la classe *X* deve effettuare l'overriding del metodo *run*, la cui intestazione in *Thread* è la seguente: *public void run()*
3) Istanziare la classe *X*
4) Invocare il metodo ***start*** dell'oggetto creato.

Naturalmente, all'occorrenza il procedimento può essere semplificato utilizzando una classe *X* anonima.

Ad esempio, creiamo un thread di esecuzione che stampa i numeri da 0 a 9, con una pausa di un secondo tra un numero e il successivo. A questo scopo, creiamo una classe che estende *Thread*, il cui metodo *run* svolge il compito prefissato:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i=0; i<10 ;i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000); // metodo bloccante
            } catch (InterruptedException e) {
                return; // chiude il thread
            }
        }
    }
}
```

- Non abbiamo dotato la classe di un costruttore in quanto ***Thread*** **ha un costruttore senza argomenti**, che è sufficiente per i nostri scopi

A questo punto, istanziamo la classe *MyThread* e facciamo partire il corrispondente thread di esecuzione:

```
MyThread t = new MyThread();
t.start(); // metodo non bloccante
```

- Osserviamo che prima di chiamare il metodo *start*, c'è un oggetto di tipo *Thread* a cui non corrisponde (ancora) nessun thread di esecuzione

- L'intestazione di *start* in *Thread* è semplicemente *public void start()*, non è necessario farne l'overriding nelle classi che estendono *Thread*
- La chiamata non è bloccante; per definizione, il nuovo thread di esecuzione svolge le sue operazioni in parallelo al resto del programma
- **Non è consentito invocare start più di una volta** sullo stesso oggetto thread, anche se la prima esecuzione del thread è terminata.
- Il nuovo thread di esecuzione esegue automaticamente il metodo ***run*** dell'oggetto thread corrispondente
    - Il metodo run viene anche detto l'entry point del thread, perché è il primo metodo che viene eseguito
    - In questo senso, il metodo main è l'entry point del thread principale
    - Quando il metodo run termina, anche il thread di esecuzione termina

Interrompere un thread

È spesso utile interrompere le operazioni di un thread, per cui ogni thread è dotato di un **flag booleano** chiamato stato di interruzione, inizialmente falso. I metodi bloccanti, come sleep e join, vengono interrotti non appena lo stato di interruzione diventa vero (per questo l'eccezione di tali metodi è verificata!).

Il seguente metodo della classe *Thread* imposta a vero lo stato di interruzione del thread sul quale è chiamato `public void interrupt()`; quindi, nonostante il suo nome, *interrupt* non ha un effetto diretto su un thread di esecuzione. In particolare, se tale thread non sta eseguendo un'operazione bloccante, la chiamata ad interrupt non ha nessun effetto immediato.

Tuttavia, la successiva chiamata bloccante troverà lo stato di interruzione a vero ed uscirà immediatamente lanciando l'apposita eccezione.

È possibile conoscere lo stato di interruzione di un thread con `public boolean isInterrupted()`
Tale metodo restituisce l'attuale stato di interruzione di questo thread, senza modificarlo.

Una applicazione dovrebbe sempre essere in grado di terminare tutti i suoi thread su richiesta; infatti, in un ambiente interattivo l'utente potrebbe richiedere la chiusura dell'applicazione in qualunque momento.
Per ottenere questo risultato, tutti i thread dovrebbero rispettare la seguente disciplina relativamente alle interruzioni:

- Se una chiamata bloccante lancia l'eccezione *InterruptedException*, il thread dovrebbe interpretarla come una **richiesta di terminazione**, e reagire assecondando la richiesta
- Se un thread non utilizza periodicamente chiamate bloccanti, dovrebbe invocare periodicamente *isInterrupted* e terminare se il risultato è vero.

Queste regole valgono soprattutto per i thread che hanno una durata potenzialmente illimitata, come quelli basati su un ciclo infinito; in questo caso, invece di usare *while(true)* è possibile utilizzare

```
while (!Thread.currentThread().isInterrupted())
```

Naturalmente, è anche possibile segnalare un'interruzione al thread in altro modo, ad esempio utilizzando lo stato di un oggetto condiviso (lo vedremo nel prossimo capitolo).

L'interfaccia Runnable

Abbiamo visto come creare thread di esecuzione istanziando sottoclassi della classe *Thread*, questo metodo ha però lo svantaggio che la sottoclasse di *Thread* che creiamo non può estendere alcuna altra classe.
In alternativa, è possibile creare un thread di esecuzione tramite una nostra classe che implementi

l'interfaccia *Runnable*, il cui contenuto è il seguente:

```
public interface Runnable {
    public void run();
}
```

Come si vede, l'interfaccia contiene solo un metodo *run* (consentendoci quindi di poter usare la lambda espressione) del tutto analogo a quello della classe *Thread*, tale metodo sarà l'entry point per il nuovo thread di esecuzione.

Per creare il thread, utilizziamo il seguente costruttore della classe *Thread*, passandogli come argomento un oggetto di una nostra classe che implementa *Runnable*: `public Thread(Runnable r)`
Naturalmente, per far partire il nuovo thread, è sempre necessario chiamare il metodo start.

È possibile usare lo stesso oggetto *Runnable* per creare più thread: tutti eseguiranno lo stesso metodo *run*.
Nota bene che quando si scrive il metodo *run* di un oggetto *Runnable* non ci si trova in una classe che estende *Thread*; quindi, non è possibile scrivere semplicemente *isInterrupted()* oppure *sleep(1000)*, ma bisogna scrivere *Thread.currentThread().isInterrupted()* e *Thread.sleep(1000)*.

19. Comunicazione tra thread e mutua esclusione

Comunicazione tra thread

Piuttosto che evolvere in maniera del tutto indipendente tra loro, è spesso utile che due thread possano **comunicare**. Siccome thread dello stesso processo condividono lo spazio di memoria, il modo più semplice per farli comunicare consiste nell'utilizzare degli **oggetti condivisi**, si tratta semplicemente di oggetti a i quali entrambi i thread posseggono un riferimento.

Supponiamo, ad esempio, di voler creare due thread che **condividano un numero intero**: un thread ne modificherà il valore, mentre l'altro ne leggerà solo il contenuto.
Supponiamo di creare due classi *MyThread1* e *MyThread2*, che estendono *Thread* e rappresentano le nostre due tipologie di thread.

Per fare in modo che condividano un numero intero si potrebbe provare con:

```
int n = 0;
Thread t1 = new MyThread1(n);
Thread t2 = new MyThread2(n);
t1.start();
t2.start();
```

- Naturalmente, **non** è possibile che i due thread, così creati, comunichino tra loro tramite la variabile *n*; difatti, essendo *n* del tipo base *int*, essa verrà passata ai due costruttore per **valore**, e non per riferimento
- Quindi, i due costruttori riceveranno entrambi zero, e, soprattutto, due copie completamente indipendenti del valore zero
- Seppure il primo thread memorizzasse e successivamente modificasse tale valore, l'operazione non avrebbe nessun effetto sulla variabile *n*, né tantomeno sul secondo thread.

Visto che il problema è dovuto al fatto che la variabile *n* è di un tipo base, si potrebbe pensare di risolvere utilizzando un *Integer* (invece di *int*). Neanche questo può funzionare poiché gli oggetti di tipo *Integer* (come tutti i tipi wrapper) sono **immutabili**; quindi, i thread non possono modificare il valore della variabile condivisa *n*. Se ad esempio un thread esegue *n++* questo non ha effetto sull'altro thread, perché quell'istruzione è equivalente (tramite l'autoboxing) a *n = Integer.valuerOf(n.intValue() + 1)*. Ovvero, viene restituito un **nuovo oggetto** *Integer*, che nulla ha che vedere col vecchio.

Dunque, per risolvere il nostro problema, ci servirebbe una classe che contenga un intero e sia modificabile. Possiamo facilmente creare una classe *MyInt*, simile ad *Integer*, ma dotata di un metodo modificatore come *setValue(int m)* (così da avere un oggetto mutabile) e in effetti, una classe del genere esiste già e si chiama *java.util.concurrent.atomic.**AtomicInteger***.

Un altro modo per far comunicare i due thread è usare una collezione fornita da Java, a partire da un semplice **array**:

```
int[] a = new int[1];
Thread t1 = new MyThread1(a);
Thread t2 = new MyThread2(a);
t1.start();
t2.start();
```

- Per quanto possa sembrare anomalo creare un array di un solo elemento, questa soluzione è corretta e rappresenta un comodo escamotage per evitare di creare un'intera classe
- Inoltre, è più comune che due thread debbano condividere un **insieme di valori** e in questi casi, sarà del tutto naturale e ragionevole utilizzare un array, o una collezione del Java Collection Framework

Sincronizzazione tra thread

Nel paragrafo precedente abbiamo ignorato i ben noti problemi di sincronizzazione che possono insorgere con l'accesso concorrente a variabili condivise. Introduciamoli attraverso un esempio di un contatore condiviso: vogliamo contare il numero di volte in cui viene eseguito il task.

```
class Task implements Runnable {
    public void run() {
        …
        counter++;
    }
}
```

```
public static void main(...) {
    Task task = new Task();
    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);
    Thread t3 = new Thread(task);
    t1.start();
    t2.start();
    t3.start();
}
```

- Ci si aspetta che alla fine del processo il nostro counter abbia il valore 3, ma questo non è detto; infatti, i tre thread potrebbero essere eseguiti in contemporanea (dipende dallo scheduler che non può essere controllato) e quindi alla fine counter sarà 1
- Praticamente l'output del programma è indicibile, possiamo solo dire che la variabile counter potrà assumere un valore tra 1, 2 e 3.
- Ovviamente se ogni thread usa un'istanza diversa di Task, counter assumerà sempre il valore 1.

Dall'esempio abbiamo capito che se due thread tentano di modificare contemporaneamente lo stesso oggetto, l'interleaving **arbitrario** stabilito dallo scheduler può far si che l'operazione lasci l'oggetto in uno stato incoerente. È necessario garantire che solo un thread alla volta possa modificare tale oggetto, questa proprietà prende il nome di **mutua esclusione** e la soluzione classica al problema prevede l'uso di **mutex**.

Modificatore synchronized

Un mutex è un semaforo binario che supporta le operazioni base di lock e unlock. Java integra i mutex nel linguaggio stesso: ad ogni oggetto, indipendentemente dal tipo, è associato un mutex (chiamato **monitor**) e una corrispondente lista di attesa.

La parola chiave ***synchronized*** permette di utilizzare implicitamente tali mutex; questo modificatore si può applicare ad un metodo, oppure ad un blocco di codice (ed a nient'altro, quindi non si può applicare ad un campo o ad una variabile).

I mutex impliciti di Java sono **rientranti** (reentrant), ciò vuol dire che un thread può acquisire lo **stesso** mutex più volte. Questo accade comunemente, ogni qual volta un metodo sincronizzato ne chiama un altro, anch'esso sincronizzato (ad esempio, ho *f* e *g* sincronizzati e dentro *f* chiamo *g* con lo stesso monitor). Se i mutex non fossero rientranti, un metodo sincronizzato che ne chiamasse un altro sullo stesso oggetto andrebbe immediatamente in deadlock.

Internamente, un mutex rientrante ricorda quante volte è stato acquisito dallo stesso thread; quindi, il mutex contiene un contatore, che viene incrementato ad ogni acquisizione (lock) e decrementato ad ogni rilascio (unlock). Il mutex risulta libero quando il contatore vale zero.

Per certi versi, un mutex rientrante è simile ad un semaforo (*counting semaphore*); tuttavia, un semaforo può essere incrementato e decrementato da diversi thread, mentre un thread non può né acquisire né rilasciare un mutex che in quel momento risulti acquisito (una o più volte) da un altro thread.

Metodi sincronizzati

Consideriamo il caso di un metodo a cui sia applicato il modificatore *synchronized*; in tal caso, diremo che il metodo è **sincronizzato**: `public synchronized int f(int n) { ... }`
Supponiamo che *x.f(3)* sia una chiamata a tale metodo. L'effetto del modificatore *synchronized* è il seguente:

- Prima di entrare nel metodo *f*, il thread corrente tenta di acquisire il mutex di *x* (fase di lock)
- Se il mutex è già impegnato, il thread viene messo in attesa che si liberi
- Quando esce dal metodo *f* (che sia per un *return* o il lancio id un'eccezione), il thread rilascia il mutex di *x* (fase di unlock)

In altre parole, quando un thread invoca un metodo sincronizzato *f* di un dato oggetto, altri thread che invochino **qualunque** metodo sincronizzato dello **stesso oggetto** devono aspettare che il primo thread esca dalla chiamata a *f*. Questo garantisce che solo un thread alla volta possa eseguire i metodi sincronizzati di ciascun oggetto.

Se un metodo **statico** di una classe *A* è sincronizzato, il thread che lo invoca acquisirà il mutex dell'oggetto **Class** corrispondente alla classe *A* (ne esiste uno per ogni oggetto di tipo Class).

In caso di overriding, un metodo che era sincronizzato può diventare non sincronizzato, e viceversa; quindi, un'interfaccia non può forzare le sue implementazioni ad avere metodi sincronizzanti. Per questo, non è *possibile applicare *synchronized* ai metodi astratti di un'interfaccia (difatti è errore in compilazione).

Blocchi sincronizzati

La parola chiave *synchronized* può anche introdurre un blocco di codice; in questo caso, parleremo di **blocco** (di codice) **sincronizzato**. Usato in questo modo, *synchronized* richiede come argomento l'oggetto del quale vogliamo acquisire il mutex, ad esempio:

```
Integer n = 0;
synchronized (n) { // sezione critica rispetto a n
    ...
}
```

Oggetto di cui voglio il monitor

- Si noti che i mutex acquisisti dai blocchi sincronizzati sono gli **stessi** che sono utilizzati anche dai metodi sincronizzati
- Quindi, se un thread sta eseguendo un blocco che è sincronizzato sull'oggetto *x*, gli altri thread devono aspettare per eseguire eventuali metodi sincronizzati di *x*.

Esempio

Supponiamo che la classe *A* abbia i seguenti metodi:

```
synchronized void f()
synchronized void g()
void h()
static synchronized void i()
```

Consideriamo due oggetti distinti *a* e *b* di tipo *A* e un thread che invoca *a.f()*
Che succede se nel frattempo un altro thread invoca…

- *a.f()* deve aspettare la fine della prima chiamata
- *b.f()* non deve aspettare poiché acquisisce il monitor di *b*
- *a.g()* deve aspettare poiché il monitor dell'oggetto *a* è occupato
- *a.h()* non deve aspettare poiché *h* non è un metodo sincronizzato
- *A.i()* non deve aspettare poiché acquisisce il monitor dell'oggetto Class

Classi thread-safe

Una classe è thread-safe se si **comporta bene** (rispetta il contratto) quando viene utilizzata da thread diversi, indipendentemente dall'interleaving dello scheduler o dall'esecuzione in runtime di questi thread, e **senza** bisogno di **sincronizzazione addizionale** o altro coordinamento da parte del chiamate.

Parafrasando, una classe thread-safe mantiene il proprio contratto anche se utilizzata da diversi thread contemporaneamente, senza sincronizzazione da parte del chiamate.

Diamone un esempio considerando la solita classe *Employee* stabilendone il seguente **invariante** di classe: "il salario è un numero non negativo". Consideriamo un metodo ***incrementSalary***, che aggiunge al salario un valore dato (anche negativo); il metodo sarà thread-safe se potrà essere invocato contemporaneamente da diversi thread, mantenendo sempre rispettato il suo contatto e l'invariante di classe.

1) Versione non thread-safe: l'invariante non è a rischio però non rispetta il contratto; infatti, non posso passargli un delta negativo. Inoltre, se chiamassi due thread con *incrementSalary(200)* potrei avere un incremento di soli 200 del salario e non del valore aspettato di 400

```
public void incrementSalary(int delta) {
    if (delta >= 0)
        salary += delta;
}
```

2) Versione non thread-safe poiché oltre al contratto di *incrementSalary* stavolta può violare anche l'invariante. Infatti, se avessi un salario di 100 e chiamassi, con due thread, *incrementSalary(-80)* potrebbe succedere che la condizione venga svolta nello stesso istante (e quindi l'if risulta vero ad entrambe) ma l'incremento no, di conseguenza avrei un salario di -60

```
public void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

3) Versione **thread-safe**:

```
public synchronized void incrementSalary(int delta) {
    if (salary + delta >= 0)
        salary += delta;
}
```

La maggior parte delle collezioni standard non è thread safe, per motivi di efficienza (farle thread safe implicherebbe un overhead di memoria non indifferente).
LinkedList, *ArrayList*, *HashSet/Map* e *TreeSet/Map* non sono thread safe.

Se più thread condividono una di queste collezioni, e almeno uno dei thread modifica la collezione (è uno "scrittore"), tutti i thread devono accedere alla collezione condivisa in mutua esclusione. Ad esempio, acquisendo il monitor di quella collezione.

20. Condition variable e code bloccanti

Le condition variable

Le condition variable (variabili di condizione) sono un classico meccanismo di sincronizzazione (un'altra scelta ai mutex), che consente ad un thread di attendere una **condizione arbitraria**, che altri thread renderanno vera.

Supponiamo che due thread condividano una variabile intera e che il primo thread debba **aspettare che il secondo ne modifichi il valore** per poter andare avanti. Una soluzione potrebbe essere quella di utilizzare una ulteriore variabile condivisa, di tipo booleano, e un ciclo del tipo: *while(!modified) { /* ciclo vuoto */ }* Questa soluzione prende il nome di **attesa attiva** ed è una soluzione "ingenua" perché durante l'attesa il thread occupa inutilmente la CPU, controllando costantemente la condizione di uscita dal ciclo.

La soluzione corretta, invece, consiste nel realizzare **attesa passiva**, utilizzando una condition variable. Una condition variable è un meccanismo di sincronizzazione che, unito ad un mutex, permette di attendere il verificarsi di una condizione tramite attesa passiva e senza rischi di **race condition**.

Le condition variable in Java

Come i mutex, anche le condition variable sono realizzate in Java in modo implicito, ovvero senza utilizzare esplicitamente oggetti di tipo "*condition variable*". In particolare, la loro funzionalità viene offerta dai seguenti metodi della classe *Object*:

```
public void wait() throws InterruptedException
```

- Intuitivamente, il metodo *wait*, chiamato su un oggetto *x*, mette il thread corrente in attesa che qualche altro thread chiami *notify* o *notifyAll* sull'oggetto *x*; quindi, l'oggetto *x* funge da tramite per permettere al secondo thread di comunicare al primo che può andare avanti nelle sue operazioni
- Come tutti i metodi bloccanti, *wait* è sensibile allo stato di interruzione del thread, e in caso di interruzione solleva l'eccezione verificata *InterruptedException*.
- Tutti e tre i metodi in esame possono essere chiamati su un oggetto *x* solo se il thread corrente possiede il monitor di *x*, in caso contrario, i metodi lanceranno un'eccezione a runtime
- Vediamo passo per passo il funzionamento interno di una chiamata del tipo *x.wait()*:
    1) Se il thread corrente non possiede il monitor di *x*, lancia un'eccezione
    2) Se lo stato di interruzione del thread corrente è vero, lancia un'eccezione
    3) In un'unica operazione atomica:
        a) mette il thread corrente nella lista di attesa di *x*
        b) rilascia il monitor di *x*
        c) sospende l'esecuzione del thread

  Se il thread viene risvegliato da *notify(All)* o da un risveglio spurio:
    4) Riacquisisce il mutex di *x*
    5) Restituisce il controllo al chiamante
- Se invece il thread viene interrotto:
    4) Riacquisisce il mutex di *x*
    5) Lancia *InterruptedException*

```
public void notify()
public void notifyAll()
```

- La differenza tra *notify* e *notifyAll* è che il primo risveglia uno solo dei thread che sono potenzialmente in attesa della condizione, mentre *notifyAll* li sveglia tutti

- Si osservi che al passo *4* (di *wait*) il thread che ha invocato *wait* tenta di riacquisire il monitor dell'oggetto *x*, ma non è detto che ci riesca subito; quindi, il thread resta bloccato al passo 4 finché il monitor non si rende disponibile.
- Funzionamento interno di *x.notify(All)*:
    1) Se il thread corrente non possiede il monitor di *x*, lancia un'eccezione
    2) Se si tratta di *notify*:
        - preleva un thread dalla coda di attesa di *x* e lo rende nuovamente eseguibile (informalmente, diremo che lo "sveglia")

       Se invece si tratta di *notifyAll*:
        - preleva **tutti** i thread dalla coda di attesa di *x* e li rende nuovamente eseguibili
    3) Restituisce il controllo al chiamante

Risveglio spurio

In casi eccezionali, il metodo *wait* può restituire il controllo al chiamante anche se non sono stati invocati i metodi *notify*/*notifyAll*; in tal caso, si parla di uno **spurious wake-up** (risveglio spurio).
Questa eventualità è stata prevista per compatibilità con alcuni sistemi operativi (come Linux), nei quali le system call che la JVM utilizza per implementare le condition variable vengono interrotte in caso di segnali.

Produttori e Consumatori

Una situazione ricorrente nella programmazione concorrente consiste nel paradigma **produttore-consumatore** (paradigma che applica le condition variable).
Si tratta di due o più thread, divisi in due categorie:

- I produttori (producer) sono fonti di informazioni destinate ai consumatori
- I consumatori (consumer) devono elaborare le informazioni fornite dai produttori, non appena queste si rendono disponibili

Poiché non è possibile prevedere quanto tempo impiega un produttore a produrre un'informazione, né quanto impiega un consumatore ad elaborarla, si pone il problema di **sincronizzare le operazioni** tra le due categorie.

La soluzione classica prevede uno o più **buffer**, che contengono le informazioni prodotte e non ancora consumate. Supponiamo che ci sia un unico buffer, con una capienza limitata.
Se un **produttore** trova il **buffer pieno** quando è pronto a produrre una nuova informazione, deve **attendere** che un consumatore liberi un posto nel buffer. Simmetricamente, se un **consumatore** trova il **buffer vuoto**, deve **attendere** che un produttore vi inserisca almeno un'informazione.

Le condition variable permettono di realizzare queste attese in modo passivo e senza rischio di race condition. Uno schema di un'implementazione in Java di questa situazione è il seguente:

- Supponiamo che il riferimento *buf* punti ad una struttura dati con metodi:
    - ***put***: aggiunge un elemento
    - ***take:*** rimuove un elemento

- I due thread seguenti usano il buffer non solo per comunicare, ma anche per sincronizzarsi:

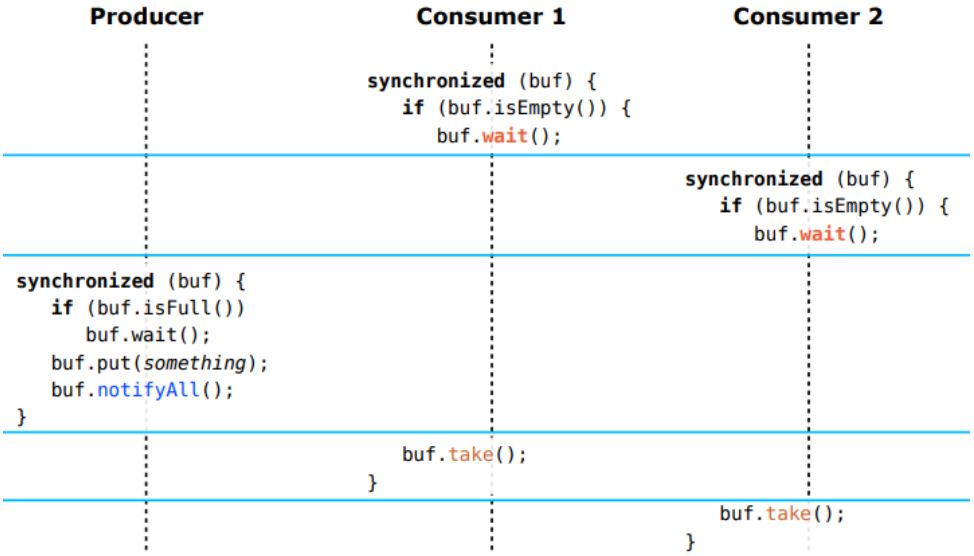
```
// PRODUTTORE:
synchronized (buf) {
    // attende che il buffer non sia pieno
    while (buf.isFull()) {
        try {
            buf.wait();
        } catch (InterruptedException e) {
            return;
        }
    }
    buf.put(some_value);
    // notifica i consumatori
    buf.notifyAll();
}
```

```
// CONSUMATORE:
synchronized (buf) {
    // attende che il buffer non sia vuoto
    while (buf.isEmpty()) {
        try {
            buf.wait();
        } catch (InterruptedException e) {
            return;
        }
    }
    some_value = buf.take();
    // notifica i produttori
    buf.notifyAll();
}
```

- Questo schema è adatto anche alla situazione con tanti produttori e tanti consumatori.

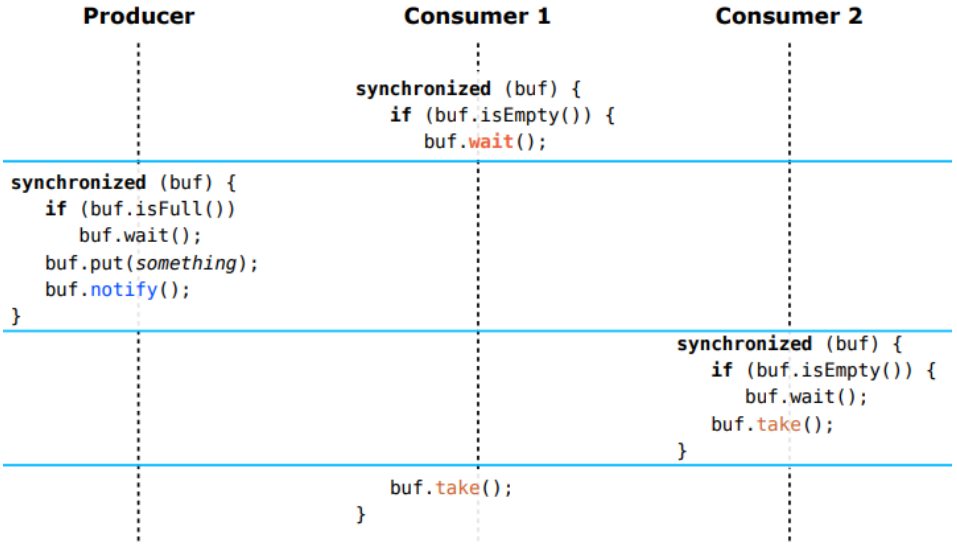
Viene naturale chiedersi perché sia il produttore sia il consumatore basano la loro attesa su di un ciclo ***while***, invece di un semplice ***if***. Ma se il consumatore (simmetrica situazione si ha con il produttore) fosse strutturato con *if (buf.isEmplty()) buf.wait();* si possono presentare **tre problemi**:

1) Se il produttore usa *notifyAll*: vengono svegliati due consumatori, ma c'è un solo valore nel buffer



Quindi con consumatori multipli e *notifyAll* può accadere una ***take* da buffer vuoto**

2) Se il produttore usa *notify*: viene svegliato un solo consumatore, ma un altro lo anticipa



Dunque, anche con solo *notify* e consumatori multipli si ha una ***take* da buffer vuoto**

3) In tutti i casi: un risveglio spurio da *wait* può portare a leggere da un buffer vuoto. Quindi, un consumatore che abbia trovato il buffer vuoto può uscire da *wait* senza un motivo specifico e tentare di leggere dal buffer vuoto. Questo problema si può risolvere soltanto **ricontrollando la condizione di attesa**, quindi usando un ciclo.

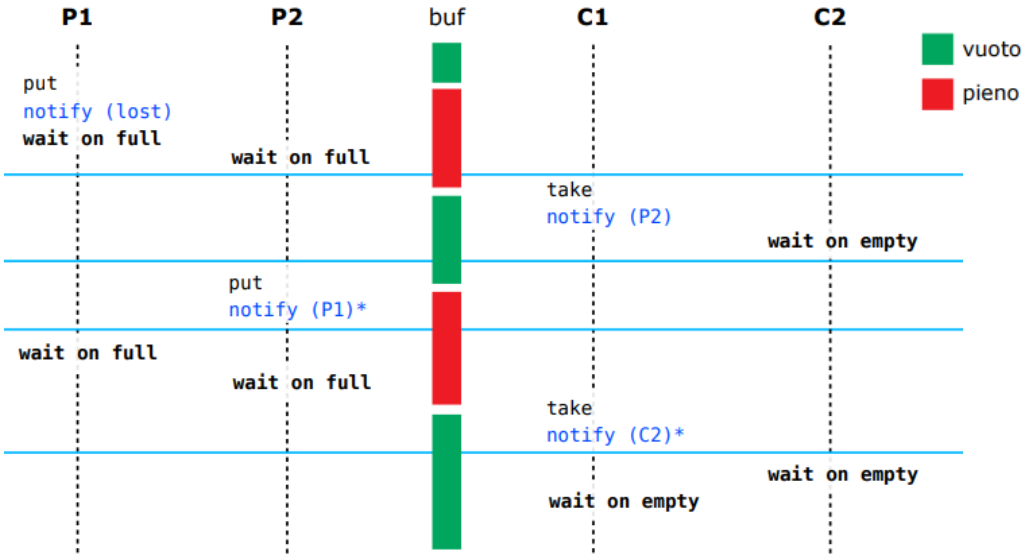
Sorprendentemente, conviene usare *notifyAll* anche se ogni prodotto è destinato ad un unico consumatore. Usare ***notify* può creare un deadlock**. Forniamo uno scenario d'esempio:

- Schematizziamo produttore e consumatore come segue:

```
// Produttore:
while (true) {
    wait on full
    put
    notify
}
```

```
// Consumatore:
while (true) {
    wait on empty
    take
    notify
}
```

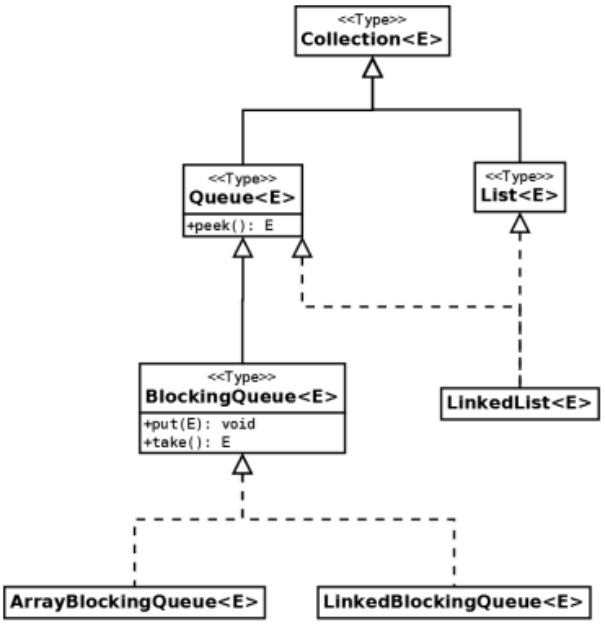
- Il buffer usato ha capienza 1 ed è condiviso tra due produttori e due consumatori
- Ogni notify è annotata con il thread che viene svegliato



- Risultato: **deadlock**

Code bloccanti

Situazioni simili al produttore-consumatore sono così frequenti che la libreria standard Java offre all'interno della Java Collection Framework delle collezioni predisposte all'utilizzo come buffer in un ambiente concorrente. Si tratta delle cosiddette **code bloccanti**, in figura si mostrano le principali interfacce e classi relative alle code bloccanti e il loro rapporto con alcune interfacce e classi già esaminate.



- Si parte dall'interfaccia parametrica ***Queue***, che rappresenta una generica coda, non necessariamente bloccante
- *Queue* viene estesa da ***BlockingQueue***, che rappresenta una generica coda bloccante
- Vengono fornite diverse implementazioni concrete di *BlockingQueue*, delle quali noi esamineremo ***ArrayBlockingQueue*** e ***LinkedBlockingQueue***

- L'interfaccia ***Queue*** aggiunge alcuni metodi a *Collection*, tra i quali menzioniamo solamente *public E **peek()***: tale metodo restituisce l'elemento in cima alla coda, senza rimuoverlo. Se la coda è vuota, viene restituito *null* (quindi *peek* non è bloccante).
    - Siccome *Queue* estende *Collection*, tutte le code dispongono dei metodi offerti da *Collection*.
    - Inoltre, *Queue* estende indirettamente *Iterable*.
- Si noti che *LinkedList* implementa, oltre a *List*, anche *Queue*, ma non *BlockingQueue*

L'interfaccia parametrica ***BlockingQueue*** rappresenta una generica coda bloccante. Essa offre dei metodi per inserire e rimuovere elementi, che si bloccano se la coda è piena o vuota, rispettivamente. Così facendo, essa permette a produttori e consumatori di sincronizzarsi senza usare esplicitamente le apposite primitive (mutex e condition variable).

L'interfaccia ***BlockingQueue*** offre, tra gli altri, i metodi *put* e *take* (che da contratto sono bloccanti):

```
public void put(E elem) throws InterruptedException
```

- Inserisce l'oggetto elem nella coda
- Le implementazioni scelgono in che posto inserirlo, tipicamente all'ultimo posto (ordine FIFO)
- Se la coda è **piena**, questo metodo mette il thread corrente in **attesa** che venga rimosso qualche elemento dalla coda
- Come tutti i metodi bloccanti, *put* è sensibile allo stato di interruzione del thread corrente, e solleva l'eccezione *InterruptedException* se tale stato diventa vero durante l'attesa, o era già vero all'inizio dell'attesa

```
public E take() throws InterruptedException
```

- Restituisce e rimuove l'elemento in cima alla coda
- Se la coda è vuota, questo metodo mette in attesa il thread corrente, finché non viene inserito almeno un elemento nella coda
- Vale per *take* lo stesso discorso fatto per *put*, in relazione allo stato di interruzione dei thread

Vediamo ora le implementazioni concrete di *BlockingQueue:*

- La classe ***ArrayBlockingQueue*** è un'implementazione di *BlockingQueue*, realizzata internamente tramite un array circolare
    - Una *ArrayBlockingQueue* ha una **capacità fissa**, dichiarata una volta per tutte al costruttore: `public ArrayBlockingQueue(int capacity)`
    - Una *ArrayBlockingQueue* rappresenta un classico buffer con capacità limitata (bounded buffer)
- La classe ***LinkedBlockingQueue*** è un'altra implementazione di BlockingQueue, realizzata tramite una lista concatenata
    - Una *LinkedBlockingQueue* ha una **capacità** potenzialmente **illimitata**
    - Quindi, una *LinkedBlockingQueue* non risulta mai piena
    - Pertanto, il metodo put di *LinkedBlockingQueue* non è bloccante
- Oltre ad essere bloccanti, queste classi sono anche **thread-safe**
- I metodi *put* e *take* di queste due classi rispettano l'ordine *FIFO*

Produttore-consumatore con coda bloccante

Le code bloccanti permettono di implementare in modo molto semplice lo schema produttore-consumatore, senza doversi occupare manualmente della sincronizzazione.

I seguenti thread condividono il buffer

```
BlockingQueue<T> buffer = new ArrayBlockingQueue<T>(capacity);
```

```java
// Produttore:
    T x;
    // "produce" x
    try {
        // attende se il buffer è pieno
        buffer.put(x);
    } catch (InterruptedException e) {
        return;
    }
```

```java
// Consumatore:
    T x;
    try {
        // attende se il buffer è vuoto
        x = buffer.take();
    } catch (InterruptedException e) {
        return;
    }
    // "consuma"
```

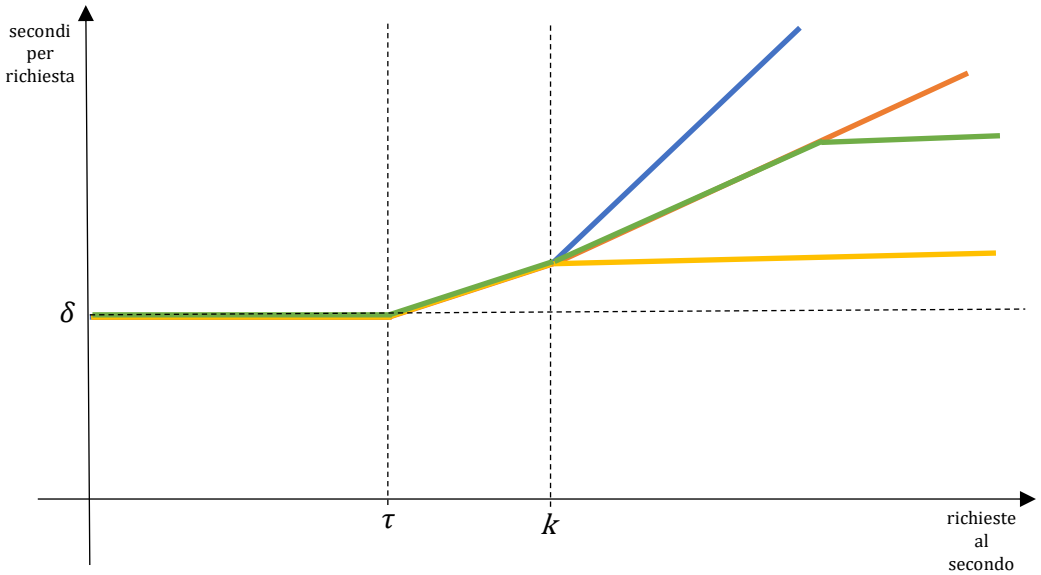
Architetture a confronto

Supponiamo di voler realizzare un **web server**:

- Il server riceve un flusso di richieste con cadenza variabile e imprevedibile
- Il server ha una capacità massima di richieste che può soddisfare al secondo

Si confrontino le seguenti architetture, al crescere del numero di richieste per unità di tempo

- **Architettura 1**: Il server crea un thread per ogni nuova richiesta, quel thread gestisce per intero la richiesta
- **Architettura 2**: Il server crea un thread per ogni nuova richiesta, con un numero massimo *k* di thread contemporaneamente attivi
- **Architettura 3**: produttore-consumatore con buffer illimitato (esempio, *LinkedBlockingQueue*)
- **Architettura 4**: produttore-consumatore con buffer limitato di capacità *k* (es., *ArrayBlockingQueue*)



δ: tempo netto per risolvere una richiesta a server "scarico"
τ: numero di richieste che il server può soddisfare in un secondo

Si noti che il precedente grafico non mostra il numero di richieste droppate dal server; infatti, bisogna tener conto delle seguenti considerazioni:

- Nell'architettura 2, una volta raggiunto il numero massimo di thread (*k*) il ritardo non cresce ma vengono droppate le richieste
- Nell'architettura 4 le richieste vengono droppate dal momento in cui finisce la coda

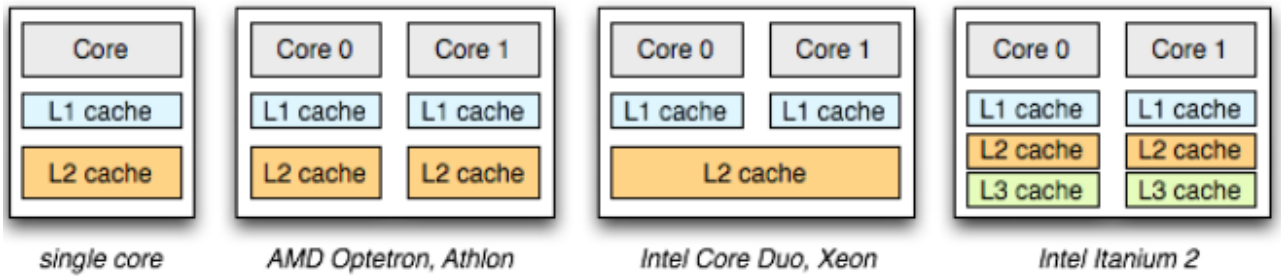
Si fa presente che in quasi tutti gli scenari concreti è meglio l'architettura 4, poiché quando il ritardo inizia a farsi consistente è comunque meglio droppare le richieste (nessuno aspetta più di 30 secondi per caricare una pagina web!).

21. Il Java Memory Model

I modelli di memoria

Un modello di memoria (**memory model**) è una descrizione di come una architettura hardware (reale o virtuale) gestisce gli accessi alla memoria. Nelle architetture moderne la memoria è **stratificata** in diversi livelli, che vanno dalla memoria di massa (hard disk), fino ai registri della CPU, passando per diversi livelli di cache; questa stratificazione prende il nome di **gerarchia della memoria** (memory hierarchy).

Alcuni livelli, come i registri e i primi livelli di cache, sono **separati** tra i diversi core, mentre altri livelli, come la RAM, sono **condivisi** tra i core. Ad esempio, questa è una parte della gerarchia della memoria in alcuni processori:



Questa stratificazione crea notevoli problemi di **sincronizzazione** tra processi (o core) diversi (problemi di questo tipo prendono il nome di "coerenza delle cache").
Il memory model presenta un **modello astratto del comportamento della memoria**, in modo da consentire agli utenti (in primo luogo, sviluppatori di compilatori, di sistemi operativi, etc…) di ragionare senza conoscere i dettagli dell'hardware. Ad esempio, il memory model specifica le regole in base alle quali una scrittura in memoria da parte di un core diventa visibile ad un altro core.

Il **Java Memory Model** (JMM) è una descrizione di come la JVM gestisce l'accesso alla memoria ed è di particolare importanza in un contesto multi-threaded, esso contiene regole di tre tipi: **atomicità** (quali operazioni sono naturalmente atomiche?)**, visibilità** (quand'è che una scrittura in memoria diventa visibile agli altri thread?) e **ordinamento** (in quale ordine vengono effettuate le operazioni?).

Il JMM è un ingrediente fondamentale della **portabilità** promessa da Java, difatti, il JMM offre alle applicazioni (e al programmatore) delle regole certe sull'accesso concorrente alla memoria.
Il compilatore e la JVM hanno il compito di conciliare il JMM con il memory model dell'architettura hardware sottostante (il JMM è stato ridefinito nel 2004 perché la versione precedente aveva problemi di vario tipo).

Regole di Atomicità

Rispondono a quali operazioni sono intrinsecamente atomiche senza bisogno di sincronizzazione, una sintassi del linguaggio che esprime garanzie su questa proprietà è il modificatore ***volatile***. Questo modificatore si può applicare esclusivamente ai **campi** di una classe.

Intuitivamente, *volatile* indica che quel campo **può essere modificato da più thread** ma tecnicamente ha conseguenze di atomicità, visibilità, e ordinamento; tutte garanzie che dà il Java memory model.

Un campo ***volatile* non può essere *final***, in quanto l'accoppiata è priva di senso (*volatile* presuppone un campo che può essere modificato da più thread, quindi se la dichiariamo *final* allora è inutile dichiararla *volatile*)

Quali operazioni sono atomiche anche in assenza di meccanismi di mutua esclusione (synchronized)?

Regole:

1) La lettura e scrittura (singola operazione, quindi o una o l'altra) di variabili di tipo primitivo, esclusi i tipi *long* e *double*, e di tipo riferimento sono operazioni atomiche
    - In un architettura a 32bit la modifica di una variabile long o double (64 bit) può avvenire in due distinte operazioni, che modificano separatamente i 32 bit più significativi e i 32 bit meno significativi.
    - Queste due operazioni potrebbero essere interrotte dallo scheduler (vanno contro dunque all'atomicità)
2) La lettura e scrittura di variabili **volatili** sono operazioni atomiche (anche long e double)

Esempi di atomicità

Date le seguenti variabili:

```
int x, y;
long n;
volatile long m;
```

Esaminiamo le seguenti assegnazioni:

1) $x = 8$; operazione atomica
2) $x = y$; operazione **non** atomica, perché comprende una lettura e una scrittura. Se l'operazione viene interrotta, un altro thread può modificare *y* ed *x* potrebbe comunque assumere il vecchio valore di *y*
3) $n = 0x1122334455667788$; (costante long espressa in esadecimale) operazione **non** atomica. Se l'operazione viene interrotta, un thread potrebbe osservare $n == 0x11223344000000$ e un altro thread $n == 0000000055667788$
4) $m = x1122334455667788$; **operazione atomica**

Aggiungiamo le seguenti variabili:

```
Object a, b;
volatile Object c, d;
```

5) $a = null$; operazione atomica
6) $a = b$; operazione **non** atomica, perché comprende una lettura e una scrittura. Stesso rischio del caso 2
7) $c = d$; operazione **non** atomica. Stesso rischio dei casi 2 e 6

Mutua Atomicità

Due operazioni (o blocchi di istruzioni) *A* e *B* sono **mutuamente atomiche** se, dal punto di vista di un thread che sta eseguendo *A*, gli effetti dell'esecuzione di *B* da parte di un altro thread vengono visti per intero, o per niente, ma **mai** "a metà" (dal punto di vista di *A*, *B* o non è ancora iniziato oppure ha completato la sue operazioni).

La mutua atomicità è una forma più debole di atomicità, **relativa** anziché assoluta; questa viene garantita dai meccanismi di locking, come il costrutto *synchronized*.
Tutti i blocchi *synchronized* sullo stesso monitor sono mutuamente atomici.

La mutua atomicità è una forma più astratta di mutua esclusione:

- Il termine **mutua esclusione** suggerisce che le due operazioni verranno eseguite in tempi diversi (non contemporaneamente)
- Il termine **mutua atomicità** parla solo di visibilità degli effetti, non di implementazione

La mutua esclusione è un modo di implementare la mutua atomicità

Regole di Visibilità

Prima di enunciare le regole diamo un concreto esempio di un problema di visibilità. Il seguente programma si comporta in un modo non deterministico:

```
static boolean done;
static int n;
public static void main(String args[]) {
    Thread t = new Thread() {
        public void run() {
            n = 42;
            try { sleep(1000); }
            catch (InterruptedException e) { return; }
            System.out.println("Fatto");
            done = true;
        }
    };
    t.start();
    while (!done) { }
    System.out.println(n);
}
```

In questo caso non c'è nessuna garanzia che il thread veda la variable *done* modificata (conseguenza indiretta della stratificazione della memoria); infatti, il programma precedente può essere schematizzato come segue:

```
                    done = false
                    n = 0
```

Thread 1 (main):	**Thread 2:**
`while (!done) { };`	`n = 42;`
`System.out.println(n);`	`sleep(1000);`
	`System.out.println("Fatto");`
	`done = true;`

- Si noti che i due thread condividono le variabili *n* e *done*, ma non usano alcun meccanismo di sincronizzazione.
- Sorprendentemente, il ciclo *while* del thread 1 può comportarsi come un ciclo infinito (provare per credere), anche se il thread 2 dopo un'attesa di un secondo esegue *done = true*.
- Questo perché, in mancanza di sincronizzazione, il JMM non offre alcuna garanzia su quando la scrittura nella variabile *done* effettuata dal thread 2 sarà **visibile** al thread 1.

Il precedente problema si spiega con il seguente principio della visibilità inter-thread:

- In mancanza di sincronizzazione, le operazioni (scritture in memoria) svolte da un thread possono rimanere nascoste agli altri thread a **tempo indefinito**. Non solo, alcune operazioni possono rimanere nascoste ed altre essere visibili.

Tutto questo caos può essere evitato se si seguono le seguenti operazioni (regole che danno garanzie di visibilità):

1) **Acquisire un monitor** (cioè, entrare in un metodo o blocco sincronizzato) rende visibili le operazioni effettuate dall'ultimo thread che possedeva quel monitor, fino al momento in cui l'ha rilasciato (nel momento in cui lascio il monitor, tutto ciò che ha fatto il thread precedente e durante l'acquisizione è visibile dal thread che acquisisce lo stesso monitor)
2) Leggere il valore di una **variabile *volatile*** rende visibili le operazioni effettuate dall'ultimo thread che ha modificato quella variabile, fino al momento in cui l'ha modificata

3) **Invocare *t.start()*** rende visibili al nuovo thread *t* tutte le operazioni effettuate dal thread chiamante, fino all'invocazione a *start*
4) **Ritornare da** una invocazione ***t.join()*** rende visibili tutte le operazioni effettuate dal thread *t* fino alla sua terminazione

3 e 4 sono delle garanzie fornite dagli effetti collaterali di *start* e *join* e quindi non ci portano a programmare in modo diverso (era evidente anche senza conoscere queste regole), invece, 1 e 2 ci forniscono degli strumenti da usare quando vogliamo ottenere garanzie di visibilità.

Infatti, applicando la regola 2 al programma precedente dichiarando la variabile *done* come *volatile*, esso avrà il comportamento atteso, perché ogni lettura della variabile *done* effettuata dal thread principale rende visibili le modifiche a *done* fatte dall'altro thread (*volatile* ha praticamente lo scopo di garantire visibilità, piuttosto delle garanzie minime di atomicità viste precedentemente).

Un confronto tra synchronized e volatile

Come si è visto, sia *synchronized* sia *volatile* offrono garanzie di **atomicità** e **visibilità**; tuttavia, il modificatore *volatile* rende atomica soltanto una singola scrittura nella variabile in questione.

Come si è visto, il modificatore *volatile* **non** rende atomica nemmeno un'assegnazione del tipo $a = b$, anche se *a* e *b* fossero entrambe *volatile*; come ulteriore esempio, il modificatore *volatile* **non** rende atomica nemmeno l'espressione *n++*. Quindi, un blocco o metodo *synchronized* rappresenta l'unica opzione per rendere atomica una sequenza di istruzioni (o comunque, per renderla mutuamente esclusiva rispetto ad altre sequenze critiche).

Il modificatore *volatile* è indicato nei casi in cui il contesto richieda la **visibilità** dei cambiamenti, **ma non l'atomicità** o la mutua esclusione.

Regole di Ordinamento

Iniziamo con un problema riguardante l'ordinamento. Si considerino i seguenti thread, che condividono due variabili *A* e *B*, inizialmente poste a ***0***:

Thread 1:	**Thread 2:**
`int r1;`	`int r2;`
`r1 = B;`	`r2 = A;`
`A = 1;`	`B = 1;`

Sembra evidente che tra tutti i valori possibili che possono assumere le variabili *r1* e r2, l'unica combinazione non possibile è 1, 1. Poiché il primo dei due che arriva alla terza istruzione avrà già fatto l'assegnazione. Sorprendentemente, il JMM consente anche questo risultato apparentemente assurdo.

Più precisamente, i valori che possono assumere alla fine le variabili *r1* e *r2* sono:

- $r1 = 0, r2 = 1$ se il *Thread 1* viene eseguito per primo
- $r1 = 1, r2 = 0$ se il *Thread 2* viene eseguito per primo
- $r1 = 0, r2 = 0$ se lo scheduler interrompe il primo thread tra le due assegnazioni
- $r1 = 1, r2 = 1$: in mancanza di sincronizzazione, al compilatore/JVM/CPU è consentito di **riordinare le istruzioni**, a patto che tale riordino sia ininfluente dal punto di vista del singolo thread

In questo caso, è consentito invertire l'ordine delle due istruzioni del *Thread 1* (oppure del *Thread 2*), poiché non cambia niente al fine del singolo thread. Quindi, se l'ordine viene invertito diventa possibile anche il risultato 1, 1.

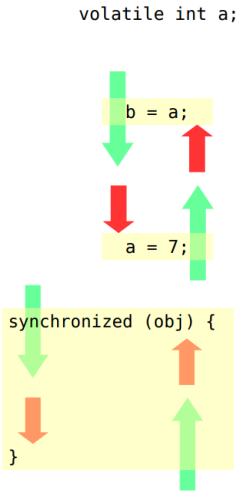
In generale, il compilatore e la JVM possono riordinare **qualsiasi sequenza di istruzioni** a patto che il risultato non cambi per un singolo thread che esegua quelle istruzioni.
I costrutti *synchronized* e *volatile* riducono le possibilità di riordino.

Distinguiamo innanzitutto tre categorie di istruzioni:

1) Letture di una variabile volatile, oppure inizio di un blocco o metodo sincronizzato
2) Scritture di una variabile volatile, oppure fine di un blocco o metodo sincronizzato
3) Tutte le altre istruzioni, che chiameremo "normali"

Consideriamo due istruzioni successive: *x; y;* (il riordino può ovviamente essere fatto solo tra due righe sequenziali) e supponiamo che **non abbiano dipendenza** dal punto di vista di un thread singolo.
La seguente tabella specifica se è possibile **scambiare** di posto le istruzioni *x* e *y*:

tipo di x \ tipo di y	normale	lettura volatile inizio synchronized	scrittura volatile fine synchronized
normale	**Si**	**Si**	No
lettura volatile inizio synchronized	No	No	No
scrittura volatile fine synchronized	**Si**	No	No



In pratica:

1) le istruzioni normali si possono sempre scambiare
2) le istruzioni normali si possono portare dentro un blocco sincronizzato
3) le istruzioni normali che precedono la lettura di una volatile si possono spostare dopo la lettura
4) le istruzioni normali che seguono la scrittura di una volatile si possono spostare prima della scrittura

Modifichiamo l'esempio dei due thread, aggiungendo un oggetto condiviso *obj* e due blocchi sincronizzati:

Thread 1:

```
int r1;
synchronized (obj) {
   r1 = B;
   A = 1;
}
```

Thread 2:

```
int r2;
synchronized (obj) {
   r2 = A;
   B = 1;
}
```

- I blocchi sincronizzati **non** impediscono alle istruzioni al loro interno di essere riordinate
- In compenso, rendono i due blocchi **mutuamente esclusivi**
- Quindi, uno dei due blocchi verrà interamente eseguito prima dell'altro
- Gli unici output possibili sono $r1 = 0, r2 = 1$ oppure $r1 = 1, r2 = 0$

In alternativa, supponiamo che *A* e *B* siano *volatile*:

Thread 1:

```
int r1;
r1 = B;
A = 1;
```

Thread 2:

```
int r2;
r2 = A;
B = 1;
```

- Consultando le regole di ordinamento (la tabella), scopriamo che il compilatore adesso **non può riordinare** le istruzioni, perché sono tutte letture o scritture di variabili *volatile*
- Gli output possibili sono: $r1 = 0, r2 = 0$ oppure $r1 = 0, r2 = 1$ o ancora $r1 = 1, r2 = 0$
- Il primo output può capitare perché *volatile* **non** rende i due thread mutuamente esclusivi

Applicazione: Lazy Inizialization

Consideriamo il problema di una classe che voglia rendere disponibile un oggetto, che sarà istanziato soltanto alla prima richiesta. Questo problema prende il nome di **lazy initialization** (inizializzazione pigra).

La soluzione naif (e non thread-safe) è la seguente:

```
class A {
    private static HeavyClass special;
    public static HeavyClass getSpecial() {
        if (special == null)
            special = new HeavyClass();
        return special;
    }
}
```

Il riferimento special sarà inizializzato con un nuovo oggetto alla prima invocazione di *getSpecial*. Ma questa implementazione soffre di due problemi diversi:

- **Scenario 1**: Due thread invocano contemporaneamente *getSpecial*
    - Il primo thread trova *special* a *null* e viene interrotto dallo scheduler. Anche il secondo thread trova *special* a *null*, quindi istanzia un oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a special ed esce da *getSpecial*.
    - Il primo thread riprende la sua esecuzione, istanzia un secondo oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a *special* ed esce anch'esso da *getSpecial*.

  Problema: sono stati istanziati due diversi oggetti *HeavyClass*, contrariamente alle intenzioni
  Commenti: questo è il classico problema dovuto alla mancata atomicità della sequenza "lettura di *special* – scrittura di *special*". Il problema non è legato alle sottigliezze del JMM

- **Scenario 2:** Due thread invocano contemporaneamente *getSpecial*
    - Il primo thread trova special a *null*, istanzia un nuovo oggetto di tipo *HeavyClass* e ne assegna l'indirizzo a *special*.
    - Il secondo thread riceve da *getSpecial* un riferimento allo stesso oggetto, accede a questo oggetto e lo trova in uno stato incoerente, cioè non completamente inizializzato dal costruttore (con i suoi campi allo stato di default).

  Problema: Il secondo thread potrebbe vedere l'oggetto a metà della sua costruzione, anche se dal punto di vista del primo thread l'oggetto è stato completamente costruito.
  Commenti: In mancanza di sincronizzazione, non vi è alcuna garanzia che il secondo thread veda tutte le operazioni svolte dal primo. Potrebbe darsi che il secondo thread veda il valore corrente di *special* (cioè, l'indirizzo del nuovo oggetto *HeavyClass*), ma non veda il valore corrente di alcuni campi di quell'oggetto. Questo problema è separato dal primo ed è legato alle regole di visibilità del JMM.

Una soluzione semplice a questo problema è dichiarare **sincronizzato** (*synchronized*) il metodo ***getSpecial***; infatti, il primo problema viene risolto rendendo mutuamente esclusive le invocazioni a *getSpecial*, mentre, il secondo problema viene risolto grazie alle garanzie di visibilità offerte da *synchronized*.

Questa soluzione impone però un **overhead di performance**, dovuto alla necessità di acquisire e rilasciare un mutex, su **tutte** le invocazioni di *getSpecial*, anche molto tempo dopo l'inizializzazione dell'oggetto *HeavyClass*, quando ormai la sincronizzazione non sarebbe più necessaria (infatti i nostri problemi sono limitati al momento in cui l'oggetto viene creato).

Nota che dichiarare la variabile *special* ***volatile*** risolverebbe solo il secondo problema ma non il primo.

Di seguito riportiamo la soluzione avanzata e più appropriata al problema della lazy initialization:

```
class A {
    private static class HeavyClassHolder {
        static HeavyClass special = new HeavyClass();
    }
    public static HeavyClass getSpecial() {
        return HeavyClassHolder.special;
    }
}
```

- Il riferimento *special* viene spostato all'interno di una classe interna statica e privata.
- La classe *HeavyClass* sarà instanziata quando il campo statico *special* verrà inizializzato.
- La virtual machine inizializza la classe *HeavyClassHolder*, e quindi il suo campo *special*, **soltanto al primo utilizzo**, cioè alla prima invocazione di *getSpecial*.
- Non ci sono problemi di sincronizzazione perché la virtual machine garantisce che **l'inizializzazione di una classe sia un'operazione atomica**.
- Quindi, questa soluzione risolve il problema della lazy initialization, senza utilizzare sincronizzazione esplicita (e chi meglio della JVM può svolgere un compito).

Software Quality: Thread Safety

Una classe è thread-safe se thread multipli possono usare i suoi oggetti simultaneamente senza sincronizzazione da parte del client.
Le classi hanno cura della sincronizzazione, quando necessario. Ad esempio, *StringBuffer* e *Vector* sono thread safe, mentre non lo sono *StringBuilder* e *ArrayList*.

Una classe non thread safe può essere causa di race condition (mancanza di sincronizzazione) o deadlock (troppa sincronizzazione indisciplinata).

La politica di concorrenza stabilisce cosa può accadere contemporaneamente e quindi deve essere thread safe, mentre il resto deve essere serializzato.
Ad esempio, presa una classe *Employee* con i metodi *getSalary* e *setSalary*, la politica di concorrenza è la seguente (prende il nome di **object-level concurrency**):

1) Metodi chiamati su **employee differenti** possono essere eseguiti in **contemporanea**
2) Metodi chiamati sullo **stesso employee** deve essere **serializzato**

Le politiche di concorrenza più comuni sono espresse nella seguente tabella:

Name	What is concurrent?	How many locks?
Class-level	Access to different classes	One lock per class
Object-level	Access to different objects	One lock per object
Method-level	Access to different methods	One lock per method of each object
Anarchy	Everything	No locks

Particolare è invece la politica di concorrenza da applicare ai water containers (visti capitolo 16):

- Se i container *a* e *b* **non** sono connessi tra loro, ogni metodo in *a* può essere eseguito in contemporanea con ogni metodo di *b*, eccetto le chiamate *a.connectTo(b)* e *b.connectTo(a)* che devono essere serializzate
- Tutti gli altri casi di invocazioni a coppie di metodi devono essere serializzati (protetti con opportuna sincronizzazione)
- Dunque, la politica di concorrenza adeguata è di un lock per **gruppo**

Riportiamo la parte di interesse dell'implementazione di Speed1:

```
// I Container hanno un singolo campo:
    Group group = new Group(this);

// Group è una classe innestata:
    static class Group {
        double amountPerContainer;
        Set<Container> members;
        ...
    }
```

Si potrebbe pensare alla seguente soluzione per *connectTo*:

```
public void connecTo(Container other) {
    synchronized (this.group) {
        synchronized (other.group) {
            ...
        }
    }
}
```

ma questo potrebbe portare ad una deadlock se due thread fanno rispettivamente le chiamate *a.connecTo(b)* e *b.connecTo(a)*.

Per prevenire i deadlock ci sono due possibili soluzioni:

1) Acquisire i monitor mentre si mantiene un **monitor globale**

```
// singolo monito per l'intera classe
static Object globalLock = new Object();

public void connecTo(Container other) {
    synchronized (globalLock) {
        synchronized (this.group) {
            synchronized (other.group) {
                ...
            }
        }
    }
}
```

- questa soluzione però cambierebbe la nostra politica definita precedentemente in una object-level
- Inoltre, ha anche problemi di race condition (non ne parleremo per questa implementazione)

2) Acquisire i monitor in un **ordine prefissato**

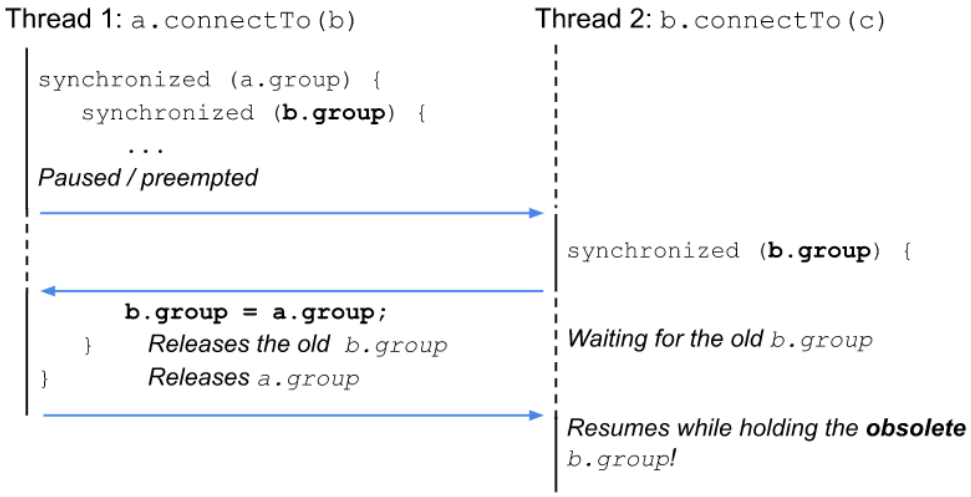
- Aggiungiamo un **unico ID** ai gruppi:

```
static class Group { // classe innestata di Container
    double amountPerContainer;
    Set<Container> elems = new HashSet<>();
    final int id = nGroups.incrementAndGet();
    static final AtomicInteger nGroups = new AtomicInteger();
                // ^ classe thread safe di interi
    ...
}
```

- Acquisiamo i monitor **nell'ordine** (crescente) **del loro ID**:

```
public void connectTo(Container other) {
    if (group == other.group) return;

    Object lock1 = group.id < other.group.id ? group : other.group,
           lock2 = group.id < other.group.id ? other.group : group;

    synchronized (lock1) {
        synchronized (lock2) {
            // merge dei due gruppi qui
        }
    }
}
```

- Anche se così il problema del deadlock è risolto rimane comunque una possibilità di race condition. Infatti, assumendo tre container non connessi *a, b, c* dove *a.group.id < b.group.id < c.group.id* potremmo avere il seguente scenario:



L'oggetto usato come monitor è cambiato (questo problema affligge anche il global monitor).

Sincronizzazione lock-free

Una soluzione al problema è quella di tentare la connessione controllando (con un ciclo while) se ci sono state "interferenze" e in quel caso riprovare.
Questo tipo di pattern ricorda l'istruzione CPU *Compare-And-Swap* (*CAS*):

```
CAS(src: address, dest: address, old: value)
```

Atomically perform the following operation:

```
If the current value of dest is old then
    swap contents of src and dest
    return true
else
    return false
```

Atomic "n++" without locks:

```
do {
   old = n
   src = old + 1
   success = CAS(src, n, old)
} while (!success)
```

Ritornando al nostro container, la soluzione ottimale che evita sia deadlock che rice condition è sfruttare un ciclo while (e quindi lock-free) al cui interno sfrutto la soluzione 2:

```
public void connecTo(Container other) {
    while (true) /* numero unbound di tentativi */ {

        Object lock1 = group.id < other.group.id ? group : other.group,
               lock2 = group.id < other.group.id ? other.group : group;

        if (lock1 == lock2) return; // già connessi

        synchronized (lock1) {
            synchronized (lock2) {
                // controllo se i lock sono attuali
                if ((lock1 == gorup && lock2 == other.group) ||
                    (lock2 == gorup && lock1 == other.group)) {
                    // merge dei due gruppi e uscita dal ciclo
                }
                // almeno uno dei due lock è obsoleto, rilascio
                // entrambi i monitor e riprovo
            }
        }
    }
}
```