Appunti di

Linguaggi di Programmazione 2

dello studente
Salvatore Bazzicalupo

June 26, 2019

Contents

Informazioni

0.1 Premessa

I seguenti appunti sono scritti seguendo le lezioni e riscrivendoli qui lezione per lezione, possono essere utili ma siccome scritti in tempi diversi potrebbero presentarsi incrongruenze, imperfezioni o errori, non mi prendo responsabilità se utilizzandoli sbagliate qualcosa.
Nonostante questo sono ben apprezzate note o correzioni, non ci metto nulla e possono rendere il file un pizzico più affidabile.
Inoltre per quanto possano essere specifici sono sempre appunti e non sostituiscono assolutamente le lezioni, le slide o i libri consigliati dal docente.
Tutti agli appunti sono riferiti all'**anno accademico 2018-2019**, per un leggero cambio di programma sono presenti meno pattern rispetto agli anni precedenti.

0.2 Libri di testo del corso di LP2

I libri consigliati dal docente sono i seguenti:

- *Core Java volume 1, edizione 10*: È un manuale di Java mirato principalmente a persone che hanno utilizzato Java solo a scopo didattico, contiene numerose spiegazioni ed esempi.
- *Java Precisely*: Anche questo è un manuale di Java ma mirato per persone che hanno già utilizzato Java in ambienti professionali, è molto più piccolo rispetto al precedente e non contiene nessuna spiegazione oltre qualche esempio.

0.3 Strumenti utilizzati

Il file è interamente scritto in LaTeX , compresi alcuni diagrammi in UML mentre per i Memory Layout ed altri diagrammi più particolari, per motivi di semplicità, è stato utilizzato il sito draw.io in quanto Tikz avrebbe richiesto troppo tempo per alcuni diagrammi.

1 Lezione 1

La lezione 1 è principalmente basata sulla presentazione del corso, degli esercizi svolti in aula per scoprire delle curiosità sui tipi di dati *float* e *double* e qualche aneddoto su cosa tratterà il corso.

1.1 Esercizi di esempio

Quesito: Dato il seguente pezzo di codice, quante volte viene eseguito il ciclo?

```
1 for(double x = 0; x != 1.0; x += 0.1) {}
```

Risposta: Il seguente ciclo non termina mai perché 1.0 non è rappresentabile tramite somma di 0.1, questa somma arriverà a $0.999\cdots99$ e poi lo supererà, quindi la condizione x != 1 sarà sempre vera.
Questo esempio ci fa capire che è importante non usare mai confronti esatti con i double come == o ! = ma bensì utilizzare condizioni che tengono conto di questo problema come >= o <=.

Quesito: Dato il seguente pezzo di codice, come si può dichiarare i in modo che il ciclo sia infinito?

```
1 while(i == i+1) {}
```

Risposta: Qui abbiamo differenti opzioni:

- `Object i = NULL;`

  Non compila, perché un riferimento non può essere incrementato siccome non esiste aritmetica dei puntatori in Java.

- `int i = Integer.MAX_VALUE;`

  Compila ma non rispetta il quesito, entra 0 volte nel ciclo in quanto sommare 1 al massimo intero rappresentante causa che esso diventi un numero negativo piccolissimo.

- `float i = Float.POSITIVE_INFINITE;`

  Funziona in quanto per lo standard IEEE: $\infty + 1 = \infty$

- `float i = Float.NEGATIVE_INFINITE;`

  Funziona in quanto per lo stesso motivo precedente.

- `float i = 1E10;`

  Anche questo funziona ma per un motivo più elaborato, un float ha 7 cifre significative, dichiarare un float molto grande, in questo caso uno con 11 cifre, ha l'effetto che la somma di piccoli valori viene persa, quindi si ha che $1E10 + 1 = 1E10$ perché la mantissa non ha abbastanza cifre per rappresentarlo e lo standard IEEE prevede che il programma ritorni il valore più vicino possibile, che è proprio 1E10.

- `double i = 1E10;`

  Funziona per lo stesso motivo precedente

Questo ci da fa capire l'importanza di tener conto della precisione del tipo di dato che stiamo utilizzando, da questi due ricaviamo la definizione di *"Code smell"*.

Definizione 1.1 (Code smell) *Si indica con "code smell" (in italiano "puzza del codice") una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione.*

1.2 Qualche nota sulle Eccezioni

Un'eccezione è un meccanismo per interrompere l'esecuzione di un sottoprogramma in caso di errore di run-time e propagare l'errore ad un livello più alto del programma stesso. In C non sono presenti eccezioni, infatti si replica una cosa simile ritornando un valore speciale.

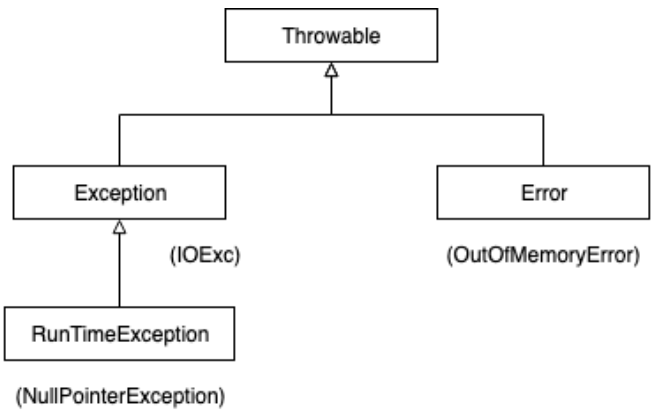
Il supporto delle eccezioni ha numerosi vantaggi come:

- Chiarezza e leggibilità di esse in quanto in Java viene esplicitato tramite una tassonomia il tipo di errore (`ArrayOutOfBondsException`, `NullPointerException`, etc), mentre in C si è necessariamente legati ai valori di ritorno, se abbiamo una funzione ritorna un puntatore, al ritorno di un puntatore null non è chiaro identificare cos'è fallito.
- Non è detto che ci possa sempre essere un valore speciale di ritorno, come una funzione che ritorna un qualsiasi valore numerico diventa impossibile usare valori speciali.
- Più sicuro in quanto un eccezione non gestita può notificare il programmatore tramite crash, mentre in C verrebbe ignorata se non curata dal programmatore e potrebbe portare a comportamenti inaspettati.
- La propagazione dell'eccezione a "ritroso" nello stack del programma

Un eccezione può essere, esclusivamente in Java, di due tipologie e sono gestite completamente dal compilatore:

- *Checked*: Viene controllata che l'eccezione sia dentro un try-catch o che il metodo possa lanciare l'eccezione tramite throws, qui è importante notare che ad esempio un `throw new IOExc()` non ha senso che sia in un try-catch bensì dovrebbe essere nel metodo.
  Vengono utilizzate quando sono esterne ed inevitabili dal chiamante, ad esempio quando un programma apre un file ma viene ritornato che esso non esiste, allora è suo compito prendersene carico.
- *Unchecked*: L'opposto delle Checked.
  Vengono utilizzate quando la condizione d'errore è causata dal chiamante e potrebbero essere evitabili.

In Java le eccezioni hanno la seguente tassonomia:



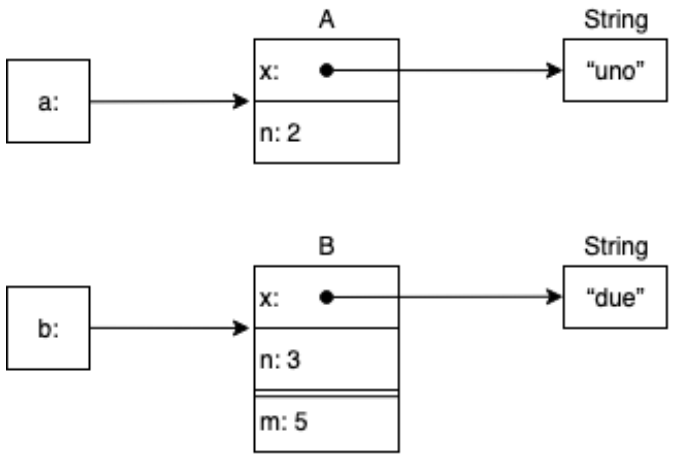
Delle seguenti Error e Runtime sono unchecked, se servono eccezioni checked si creano di tipo Exception mentre unchecked di tipo Runtime, solitamente non si devono creare nuovi Error.

1.3 Memory Layout

Basato sull'Object diagram e modificato serve per capire la situazione in memoria.
Preso ad esempio il seguente codice:

```
Class A {
   private String x;
   private int n;
   //Costruttore standard
   public A(String x, int n) {
      this.x = x;
      this.n = n;
   }
}

Class B extends A {
   private int m;
   public B() {
      super("due", 3);
      m = 5;
   }
}

public static void main(String args[]) {
   A a = new A("uno", 2);
   B b = new B();
}
```

Il suo memory layout avrà la seguente forma:



Questo diagramma si può visualizzare come: *a, b* sono gli elementi del Main che si riferiscono alle rispettive classi *A, B,* dentro di loro c'è un attributo di tipo String, che è un'altra classe, quindi quello punta alla stringa mentre gli interi che sono tipi primitivi sono dentro la classe. È importante notare che la doppia linea in B divide gli attributi "ereditati" da A ed i suoi attributi.

2 Lezione 2

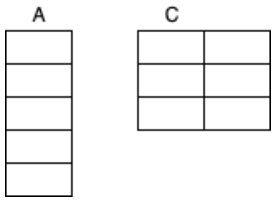
Nella lezione 2 si rispolverano alcuni argomenti di *Linguaggi di Programmazione I* e si approfondisce il sistema dei tipi.

2.1 Memory Layout con gli array

Siano dato il seguente codice in C:

```
1 int a[5];
2 int c[3][2];
```

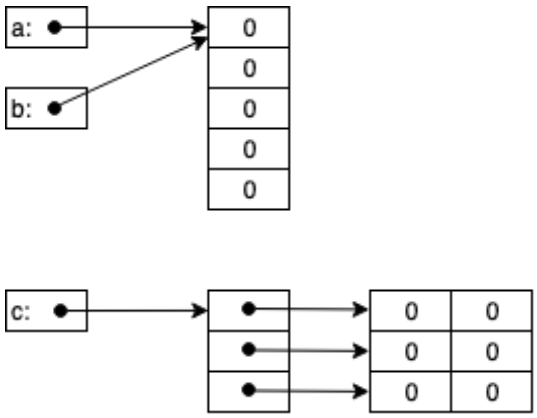
Il suo memory layout corrisponde a:



Ora sia il seguente codice in Java, simile al precedente:

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

Avremo il seguente memory layout:

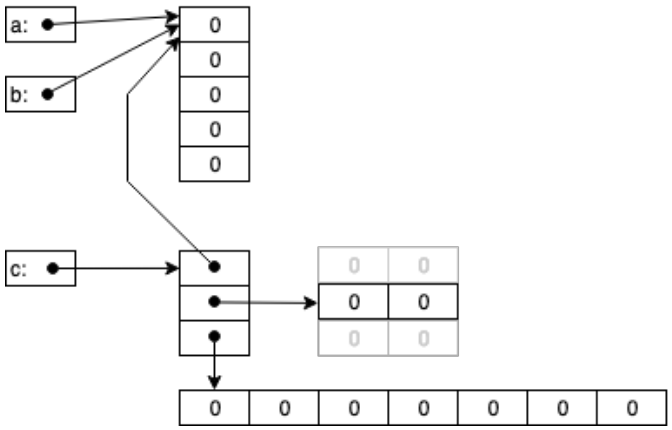


Possiamo notare subito alcune differenze, tra le quali in Java gli array bidimensionali sono semplicemente array di array, inoltre sono tutti inizializzati a 0 rispetto al C che contiene valori sporchi.

Adesso sia il seguente codice in Java, simile al precedente ma con l'aggiunta di due nuove righe:

```
1 int [] a = new int[5];
2 int [][] c = new int[3][2];
3 int [] b = a;
4 c[0] = b;
5 c[2] = new int[7];
```

Avremo il seguente memory layout:



Analizziamo cosa succede:

- Nella riga 4 abbiamo che il riferimento nella cella 0 dell'array viene cambiato in modo che punti all'array a, siccome b punta ad esso.
- Nella riga 5 nella cella 2 viene dichiarato un nuovo array di 7 elementi, di conseguenza questa cella punterà ad esso.
- Infine abbiamo 4 celle che non sono più puntate da nulla, rappresentate in grigio, che sono elegibili per il garbage collector.

2.2 Constructor chaining

La concatenazione di costruttori è il processo nel quale un costruttore ne richiama un altro che sia suo oppure uno della sua superclasse.

Per il seguente argomento prenderemo in considerazione l'esercizio del *27/03/08 n.4*: sia dato il seguente codice in Java della classe A, costruire una classe B in modo che compili con successo, non importa se poi lancia eccezioni in runtime, ma che abbia successo in compile time.

```
1  public class A extends B {
2     public A(int x) {
3        super(x-1, x/2.0);
4     }
5
6     public A(double y) {}
7
8     private void stampa(String s) {
9        if(s == null)
10          throw new B(s);
11       else
12          System.out.println(s);
13    }
14 }
```

Procediamo con la costruzione della classe B tenendo conto delle seguenti osservazioni:

1. Alla riga 3 viene richiamato il costruttore della superclasse che ha come parametri un intero ed un double (Si noti che 2.0 non ha una f vicino).
2. Alla riga 6 viene richiamato in modo implicito il costruttore vuoto di B, siccome abbiamo già un costruttore per la riga precedente è obbligatorio metterlo o il codice non compilerebbe.
3. Alla riga 10 avvengono 2 cose, viene lanciata un eccezione unchecked (siccome non c'è né un `try-catch` né A è dichiarato con `throws`) e viene lanciato il costruttore di una stringa.

Concludiamo quindi con la seguente classe B:

```
1  public class B extends RuntimeException { //Punto 3: Unchecked Exception
2    public B(int n, double z) {} //Punto 1
3
4    public B() {} //Punto 2
5
6    public B(String s) {} //Punto 3: Costruttore con stringa
7  }
```

2.3 Sistema dei tipi

Eccetto l'assembly tutti i linguaggi tipati hanno determinati tipi che aiutano il compilatore a rilevare possibili errori e notificarli, questo viene svolto durante la fase di *Type Checking*.

Vediamo le fasi del *Type Checking* con il seguente codice in Java:

```
1  int n;
2  double d;
3  d = (new Object()).hashCode() + n/2.0;
```

Il type checking dell'espressione alla riga 3 avviene dall'operatore più interno a quello più esterno, i punti nel dettaglio sono i seguenti:

1. Rileva che c'è un assegnamento per '`=`'
2. Il compilatore procede a controllare che il risultato dell'espressione destra sia compatibile con quella sinistra
3. Rileva la somma per il '`+`' e inizia a creare un albero, con nodi le due parti separate dal '`+`'
4. Rileva la divisione '`/`' a destra ed il '`.`' a sinistra
5. Da qui arriva alla costante destra e la analizza: 2.0 è un double, poi va a cercare il valore di `n` e scopre che è un int, il loro risultato è un double
6. Poi procede con l'espressione sinistra che ha l'operando '`.`', da cui va a sinistra e trova un `Object` dopodiché stabilisce se il metodo `hashCode` è presente nella classe `Object`, se è visibile, se corrisponde coi parametri e così via, infine da un valore a tutta l'espressione che è quello di ritorno del metodo: un int
7. Ora valuta l'operatore più esterno, ovvero '`+`', che è int + double
8. Il risultato della somma è un double e verifica se anche `d` è un double
9. Siccome anche l'operazione di assegnamento è lecita, questa espressione supera il type checking

Questa operazione è anche detta *parsing*, dove da una determinata espressione si costruisce un albero.

In basso è possibile vedere l'ordine di rilevamento del compilatore (1, 2, 3, 4), l'ordine di esecuzione delle istruzioni è al contrario (4, 3, 2, 1).

$$d \underbrace{=}_{1} \underbrace{(\underbrace{new\ Object()}_{4})}_{3} \underbrace{.}_{3} hashCode() \underbrace{+}_{2} n \underbrace{/}_{3} 2.0;$$

2.4 Data types

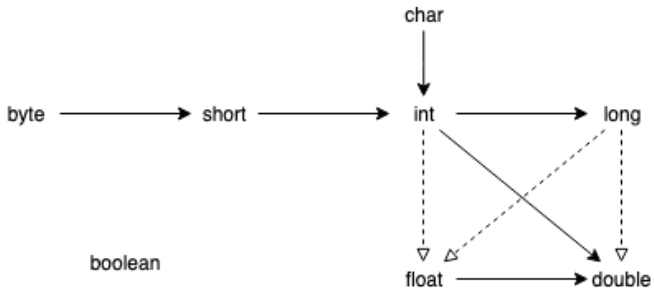
In Java abbiamo 4 tipologie di tipi:

1. Primitivi
2. Riferimenti
3. Riferimenti (Array)
4. Nullo

2.4.1 Le promozioni dei tipi primitivi

I tipi primitivi corrispondono ai tipi classici del C, ad esempio `int`, `float` o `double`, da non confondere con `Integer`, `Float` e `Double` che sono i loro rispettivi *Wrappers*.

I tipi primitivi utilizzano un sistema di promozioni per il quale possono essere convertiti in modo implicito secondo questo schema:



Le frecce nere indicano conversioni implicite mentre le frecce tratteggiate indicano conversioni consentite ma pericolose in quanto è presente una perdita di informazioni, ad esempio un `int` è 32 bit così come un `float` è 32 bit, però si noti che in un `int` tutti i bit sono dedicati alla mantissa, mentre per il `float` ne sono dedicati solo 24, avendo meno cifre significative è presente una perdita di informazioni.

Inoltre è possibile convertire anche in senso opposto alla freccia, ma questo richiede un **cast esplicito**.

Si noti inoltre che il tipo `boolean` non ha frecce, quindi nessun tipo può essere convertito in `boolean` ed il tipo stesso non può essere convertito in altri tipi, questo è dovuto al fatto che il tipo `boolean` è rappresentato con un singolo bit: 1 se *true*, 0 se *false*.

2.4.2 Regole dei tipi non primitivi

Per i tipi non primitivi invece è presente una relazione di sottotipo:
$\forall$ tipo S, T non primitivo :

1. S è un sottotipo di `Object`
2. S estende o implementa T ovvero S è un sottotipo di T
3. Se S è un sottotipo di T allora $S[\,]$ è sottotipo di $T[\,]$
4. Il tipo nullo è sottotipo di S
5. S è sottotipo di S

Siccome questa relazione rispetta la *Riflessività* (5), *Transitività* (1...5) e *Antisimmetria* (1...5) possiamo dire che questa è una **relazione d'ordine** ed è **parziale** in quanto prese 2 classi disgiunte non sono confrontabili tra di loro (Si pensi ad Exception ed Error).

È importante notare che anche i tipi primitivi rispettano una simile relazione d'ordine, ovvero:
$\forall$ tipo S, T : S è assegnabile a T

1. S, T primitivi ed esiste un assegnazione (o percorso) da S a T oppure
2. S è sottotipo di T

2.4.3 Conseguenza delle regole dei tipi

Quesito: Dato il seguente pezzo di codice dire se compila e se affermativo come si comporta in runtime.

```
1 A[] arr = new A[10];
2 Object[] arr2 = arr;
3 arr2[5] = new Object();
4 A a = arr[5];
```

Compila? Verifichiamo riga per riga:

1. Banalmente compila, dichiaro un array.
2. La regola 3 afferma che siccome `A` è un sottotipo di `Object` quindi compila.
3. Sempre per le regole di tipo è consentita l'assegnazione.
4. Ancora valida per le stesse regole.

Come si comporta in runtime? Verifichiamo riga per riga:

1. Inizializza un array di `null`.
2. `arr2` punta ad `arr` ed è consentito in quanto `A` è sottotipo di `Object`.
3. Si noti che `arr2` è un array di `Object` che punta ad un array di `A` (In java gli array memorizzano il loro tipo e la loro dimensione), siccome ad un array di `A` viene assegnato un `Object` si solleva l'eccezione `ArrayStoreException`.

2.4.4 Tipo statico e dinamico di una variabile

Sia `a` una variabile di tipo riferimento, distinguiamo due tipologie di `a`: il suo tipo statico ed il suo tipo dinamico, ovvero è di tipo statico riferimento mentre è di tipo dinamico il tipo di valore a cui punta.

Tornando al type checking quindi possiamo dire che ha successo se il tipo statico del risultato dell'espressione coincide con il tipo statico di `a`.

`a = exp` ha successo $\iff$ Tipo(`a`) = Tipo(`exp`)

2.4.5 instanceof

L'operatore `instanceof` ha la seguente struttura:
`<exp> instanceof <type>`
Indicando con `<exp>` un espressione e con `<type>` un tipo non primitivo
Esempio: `x.f() instanceof String`

L'operatore `instanceof` restituisce *true* se e solo se il tipo di `<exp>` è un sottotipo di `<type>` ed è diverso dal tipo nullo.

2.4.6 Coercizione di tipi (Cast)

I cast tra i tipi possono essere i seguenti:

- Upcast, ovvero un tipo è promosso da uno basso ad uno alto, questo viene fatto implicitamente se necessario.
- Downcast, ovvero un tipo è degradato ad uno minore, deve essere fatto esplicitamente.

Si prenda il seguente codice di esempio in Java, sui tipi primitivi:

```
1 double x = (double) 7; //Upcast, ridondante ma consentito
2 double x = 7; //Upcast implicito
3 int x = (int) 7.5; //Downcast esplicito con perdita di informazioni
4 int x = (int) true; //Errore di compilazione, non esistono frecce verso o dal tipo
    boolean ad int
```

Invece per i riferimenti si applica un concetto diverso, si noti il seguente codice in Java:

```
1 Object x = (Object) "ciao" //Upcast da String ad Object, ridondante ma consentito
2 String s = (String) new Object(); //Downcast consentito ma lancia l'eccezione a
    runtime "ClassCastException"
```

Verrebbe da chiedersi perché il compilatore consenta la riga 2 e non notifichi di errore, ma questo è dovuto per consentire l'utilizzo dei *Generics*, che verranno spiegati successivamente, infatti prima di effettuare un assegnamento di una classe specifica di un valore generico, è importante utilizzare `instanceof` per verificare che il tipo sia compatibile, si noti il seguente codice che mostra un downcast corretto:

```
1 f(Object x) {
2   if(x instanceof String) {
3      String s = (String) x;
```

```
4       System.out.println(s.length());
5     }
6   }
```

Infine è importante notare che alcuni downcast possono essere bloccati anche in compilazione se il downcast non può essere mai possibile, si noti il seguente esempio:

```
1   String s = (String) new Integer;
```

Questo è dovuto al fatto che un Integer non potrà mai essere una stringa durante l'esecuzione del programma, sono tipi incompatibili e non confrontabili tra di loro mentre precedentemente un `Object` potrebbe puntare ad una Stringa in un determinato momento ed è consentito in compile time.

3 Lezione 3

La lezione 3 si concentra pesantemente sul definire le regole del dynamic bindings e sulle conseguenze che hai con i Wrappers.

3.1 Dynamic binding (overloading e overriding)

Il dynamic binding si divide in due tipi:

- *Overloading*: "Sovraccaricamento" di una funzione, dove viene dato il medesimo nome ma cambiano gli argomenti in ingresso.
- *Overriding*: "Sovrascrittura" di una funzione, dove un funzione della superclasse viene sovrascritta dalla sottoclasse.

Quesito: Dato il seguente pezzo di codice dire quale metodo della classe **A** sarà chiamato? Il primo, il secondo o si verificherà un errore di compilazione?

```
1 public class A {
2   public void f(double x, long y) {}
3   public void f(int x, double y) {}
4 }
5
6 public static void main(String args[]) {
7   A a = new A();
8   a.f(1, 2);
9 }
```

Risposta: Errore di compilazione, il motivo sarà spiegato a breve entrando nel dettaglio di come viene eseguito il dynamic binding in Java.

3.1.1 Le regole del dynamic binding

Il dynamic binding in Java è gestito, brevemente, nel seguente metodo:

1. *Early binding*: Viene eseguito dal compilatore e si occupa dell'overloading

1.1. Identificazione delle firme candidate
1.2. Decisione della firma più specifica tra le candidate
1.3. In caso di fallimento viene eseguito di nuovo il punto (1.1) attivando l'auto-boxing

2. *Late binding*: Viene eseguito dalla JVM (Java Virtual Machine) in runtime e si occupa dell'overriding

Entriamo nel dettaglio riguardo l'early binding:

1. *Early binding* (1.1): Il compilatore va a "pescare" le firme candidate a partire dal tipo del riferimento, va nella classe e cerca tutte le espressioni *accessibili* e *compatibili*, se queste due proprietà sono rispettate allora va nell'elenco delle firme candidate, se il compilatore non trova nessuna firma candidata segnala l'errore: "no suitable method found"

2. *Early binding* (1.2): È il punto che causa l'errore di compilazione dell'esercizio precedente, perché in questo punto il compilatore cerca tra le firme candidate quella più specifica delle altre, se non ha successo allora segnala l'errore: "reference is ambiguous"
3. *Early binding* (1.3): Questo è specificato nel capitolo 3.2.2
4. *Late binding*: Si occupa di andare a riprendere la firma assegnata precedentemente dal compilatore in base al tipo dinamico che avrà la variabile in runtime, quindi potrebbe andare a prendere un metodo sovrascritto o dalla superclasse, questa fase non può fallire in quanto se ha avuto successo l'early binding il metodo esiste.
   Si noti che se il metodo è `Final`, `Static` o `Private` (ovvero il metodo non può essere sovrascritto) il late binding non avviene in quanto tutto viene fatto direttamente dal compilatore.

Diamo di conseguenza le seguenti definizioni:

Definizione 3.1 (Firma candidata) *Si definiscono firme candidate tutte le espressioni che rispettano le seguenti proprietà:*

- *L'espressione è* ***accessibile****: ovvero è visibile dalla classe che effettua la chiamata, questo riguarda specialmente i modificatori di visibilità.*
- *L'esperessione è* ***compatibile****: ovvero hanno stesso nome, stesso numero di parametri formali ma che possono avere tipo diverso purché rispettino le regole di assegnabilità.*

Definizione 3.2 (Specificità) *Prese due firme candidate:*

1. $f(T_1, T_2, \ldots, T_n)$
2. $f(V_1, V_2, \ldots, V_n)$

Diremo che 1 è più specifica di 2 se $\forall i \in 1...n : T_i$ *è assegnabile a* V_i

3.1.2 Esempio di dynamic binding

Dato il seguente esercizio della traccia d'esame del *08/09/09 n.2*, stabilire l'output e l'elenco delle firme candidate per ogni chiamata a metodo (escluso `System.out.println`):

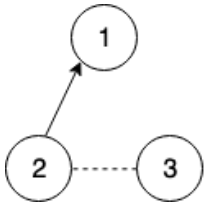
```
Class A {
   public String f(double d, A x) { return "A1"; }
   public String f(double d, B x) { return "A2"; }
   public String f(int n, Object x) { return "A3"; }
}
Class B extends A {
   public String f(double n, B x) { return "B1"; }
   public String f(float n, Object x) { return "B2"; }
}
Class C extends A {
   public String f(int n, Object x) { return "C1"; }
}

public static void main(String args[]) {
   C gamma = new C();
```

```
16      B beta = new B();
17      A alpha = beta;
18
19      System.out.println(alpha.f(3, beta));
20      System.out.println(alpha.f(3.0, beta));
21      System.out.println(beta.f(3.0, alpha));
22  }
```

Partendo dalla riga 19, cerchiamo le firme candidate, sappiamo che `alpha` è di tipo `A`, quindi dobbiamo vedere tutte le firme accessibili e compatibili, l'accessibilità in questo caso non è un problema in quanto le firme sono tutte pubbliche, valutiamo quindi la compatibilità, il primo parametro è una costante ed in Java è considerata un `int` e `beta` è un `B`, quindi la lista delle firme candidate sarà la seguente:

1. `f(double n, A x);`
2. `f(double n, B x);`
3. `f(int n, Object x);`

Ora si passa alla scelta del metodo più specifico, per fare ciò si usa un metodo grafico, per ogni firma candidata trovata disegniamo un cerchio e procediamo a fare vari confronti:



Confrontiamo 2 e 1: Qui vediamo che entrambi hanno un `double` come primo parametro ma il 2 ha un `B` come secondo parametro, che coincide con la chiamata a metodo del main, quindi stabiliamo che 2 è più specifico di 1, facciamo una freccia che parte da 2 a 1 per indicare questo confronto concluso con 2 "vincente".

Confrontiamo 2 e 3: Qui invece incappiamo nell'ambiguità in quanto il 2 ha un punto di specificità (`B` più specifico di `Object`) ma anche 3 ha un punto di specificità (`int` più specifico di `double`)
Conclusione: Concludiamo senza confrontare 1 e 3 in quanto 1 è "sconfitto" da 2 ma 2 e 3 sono ambigui fra loro, questa chiamata porta ad un errore di compilazione.

Con lo stesso principio risolviamo le altre due chiamate:

1. L'output della riga 20 è B1 e le firme candidate trovate sono:

   - `f(double n, A x);`
   - `f(double n, B x);`

   Come visto precedentemente la seconda è più precisa della prima.

2. L'output della riga 21 è A1 e le firme candidate trovate sono:

   - `f(double n, A x);`

In `B` non vengono trovate firme candidate, salendo nella superclasse se ne trova una, siccome unica firma candidata, allora è automaticamente la più specifica, si noti che `f(double n, B x);` della classe `B` non è candidato perché il compilatore non sa che `alpha` punta ad un `B` in runtime.

3.2 Wrappers

Sono classi che "racchiudono" tipi primitivi, esse sono necessarie principalmente per i *Generics* in quanto non accettano tipi primitivi, tutti i tipi numerici estendono una classe `Number` da cui ereditano metodi di conversione come `intValue`, `doubleValue`, etc.

3.2.1 Wrappers, auto-boxing e auto-unboxing

L'auto-boxing è una funzionalità di Java che converte in modo implicito tipi primitivi e Wrappers come ad esempio `int` in `Integer`, si noti il seguente codice in Java:

```
1 Integer n = 7;
2 Integer m = new Integer(7);
```

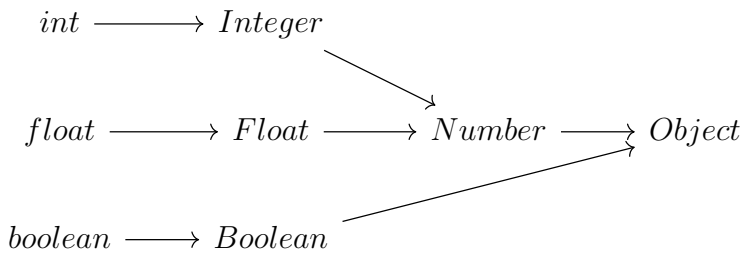
Sono entrambi metodi di dichiarazione lecita, ma si noti che è nella prima riga che si verifica l'auto-boxing, il compilatore traduce quella riga come `Integer n = Integer.valueOf(7)`.
Nonostante siano entrambi metodi validi è importante notare che l'auto-boxing sta riciclando un oggetto già esistente a differenza della seconda riga dove il `new` garantisce un oggetto nuovo, questo è dovuto al fatto che gli `Integer`, come le `String`, in Java sono immutabili.
La funzione `valueOf()` utilizza un sistema di caching e tiene memoria esclusivamente interi fino a 255 in quanto memorizzarne di più grandi causerebbe un overhead.

È importante notare alcuni errori pericolosi che possono essere causati dall'auto-boxing, si prenda il seguente codice di esempio:

```
1  Integer a = new Integer(7);
2  Integer b = new Integer(7);
3  Integer c = 7;
4  Integer d = 7;
5  Integer e = 700;
6  Integer f = 700;
7
8  System.out.println(a == b); //FALSE: a e b sono oggetti differenti!
9  System.out.println(c == d); //TRUE: La cache li memorizzano e quindi sono lo stesso
       oggetto
10 System.out.println(e == f); //FALSE: La cache non memorizza valori cosi' grandi e li
       crea come oggetti distinti
```

La morale è di non confrontare i Wrappers utilizzando == in quanto può generare comportamenti inaspettati del programma, si utilizza bensì `equals()`, anche questo può essere considerato un code smell.

Le conversioni dell'auto-boxing sono le seguenti:



Si noti che non esiste conversione implicita (o promozione) per i Wrappers, si vedano i seguenti esempi:

```
Double d = 1; //Errore di compilazione: non si puo' convertire int in Double
Double e = 1.0f; //Errore di compilazione: non si puo' convertire float in Double
Integer n = (byte) 1; //Errore di compilazione: nonostante 1 sia un int viene
    castato a byte e di conseguenza fallisce l'assegnazione
```

Riguardo l'auto-unboxing è più permissivo in quanto è possibile assegnare Wrappers a tipi primitivi seguendo lo schema iniziale di promozione automatica, si noti il segunente esempio:

```
Integer n = 7;
double d = n; //Consentito
long x = new Integer(6); //Consentito
```

3.2.2 Wrappers ed early binding

Accennato prima, ma lasciato incompleto, l'early binding per motivi di retrocompatibilità ha una terza fase, dopo aver trovato una lista vuota prova ad eseguire nuovamente la ricerca dei candidati cercando anche con i *Wrappers*, se fallisce anche questo tentativo ritorna errore.

Per concludere il dynamic binding si vedano i seguenti esempi che tengono conto anche di questo nuovo passaggio.
Esempio 1:

```
int foo(int x, Object o);
int foo(long x, String s);
```

Analizziamo i seguenti casi:

1. `foo(new Integer(7), "ciao");`

1.1. Il primo tentativo da una lista vuota, non esiste nessuna firma candidata in quanto `Integer` non è compatibile né con `int` né con `long`.

1.2. Il secondo tentativo restituisce invece entrambi i metodi, poi procede a cercare quello specifico come già mostrato, però da errore di compilazione in quanto sono equalmente specifici e di conseguenza ambigui.

2. `foo(new Float(7f), "ciao");`

2.1. Lista vuota.

2.2. Lista vuota, errore di compilazione.

3. `foo(new Long(7), "ciao");`

3.1. Lista vuota.

3.2. Ha successo e restituisce il secondo metodo in quanto quello più specifico.

Esempio 2:

```
1 int foo(double x, Integer i);
2 int foo(Double x, Integer i);
```

Analizziamo i seguenti casi:

1. `foo(1.0, 7);`

1.1. Lista vuota.

1.2. Entrambe candidate ma nella scelta della più specifica valgono le regole classiche, quindi `double` e `Double` sono la stessa cosa e si verifica errore di ambiguità.

2. `foo(1, 7);`

2.1. Lista vuota.

2.2. Restituisce la prima in quanto per 3.2.1 non esiste conversione implicita dei Wrappers, quindi l'ipotetico assegnamento `Double = 1` non è lecito.

4 Lezione 4

La lezione 4 si concentra sulla differenza di uguaglianza di indirizzo ed uguaglianza strutturale, sul metodo Equals e le sue proprietà.

4.1 Uguaglianza tra oggetti

Verificare un uguaglianza è particolarmente importante nei linguaggi ad oggetti, questo può essere fatto in due modi distinti in base al tipo di un espressione:

- Se *primitivo* allora si utilizza l'operatore `==`
- Se è un *oggetto* allora si può utilizzare sia l'operatore `==` che il metodo `equals`
  - L'operatore `==` verifica l'uguaglianza di indirizzo, ovvero se le due espressioni danno come risultato lo stesso oggetto in memoria oppure no
  - Il metodo `Equals` verifica l'uguaglianza strutturale, ovvero se il contenuto dei due oggetti è uguale oppure no

Il metodo `Equals` è un metodo della classe `Object` ed ha come implementazione base `==`, in Java ha la seguente intestazione:

```
1 public boolean equals(Object obj)
```

Il vero scopo di `Equals` è di essere ridefinito nelle varie classi per trasformarlo in un effettiva uguaglianza strutturale, si prenda il seguente esempio, commentato in dettaglio successivamente:

```
1  class Employee {
2    private String name;
3    private int salary;
4
5    @override
6    public boolean equals(Object other) {
7       if(!(other instanceof Employee) { return false; }
8
9       Employee e = (Employee) other;
10      return name.equals(e.name) && salary == e.salary;
11   }
12 }
```

Analizziamo le righe più importanti:

- Nella riga 5 è presente l'annotazione opzionale `@override`, questa permette di associare metadati ad elementi del programma ed indica al compilatore che quello che stiamo per fare è un override, questa aiuta in quanto il compilatore segnala errori se la firma non coincide con una firma della superclasse
- Nella riga 6 si verifica l'override, per quanto precisato sopra se si sostituisse `Object` con `Employee` il compilatore segnalerebbe errore
- Nelle righe 7 e 8 verifico che l'oggetto sia di tipo `Employee` prima di verificare l'uguaglianza strutturale, se il tipo non coincide ritorno direttamente `false`

- Nella riga 10 faccio il cast di `Object` in `Employee`, se sono arrivato qui allora `instanceof` mi ha ritornato che il tipo è compatibile, inoltre questo cast è necessario in quanto se non lo effettuassi non potrei accedere ai campi `name` e `salary` di `other`
- Nella riga 11 ritorno il confronto tra `name` e `salary`, si noti che il primo viene effettuato con `equals` in quanto `String` è un oggetto mentre il secondo viene confrontato con `==` in quanto tipo primitivo.

4.1.1 Proprietà di equals

Affinché il metodo `equals` sia corretto deve rispettare una *relazione di equivalenza*, questo vincolo è necessario in quanto altrimenti non funzionerebbero altri metodi come `hashCode` ed altre strutture come collezioni e mappe in quanto dipendono fortemente da esso.

Ricordiamo le proprietà algebriche di una relazione di equivalenza:

1. Riflessività: $\forall x : x.equals(x)$
2. Simmetria: $\forall x, y : x.equals(y) \Rightarrow y.equals(x)$
3. Transitività: $\forall x, y, z : x.equals(y) \ \wedge \ y.equals(z) \Rightarrow x.equals(z)$

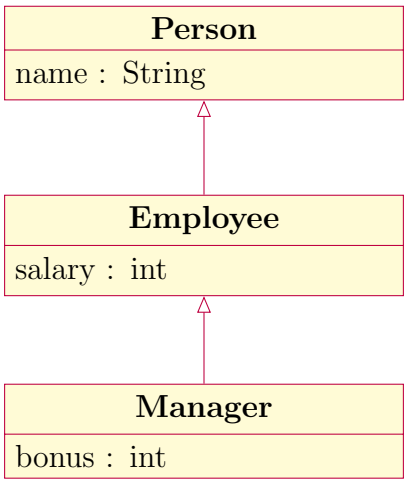
Si noti che si è utilizzata la notazione che si usa in Logica, ovvero $x.equals(x) \iff x.equals(x) = true$, di conseguenza vero è implicito mentre falso è sempre esplicitato.

Prendiamo alcuni esempi di validità di `equals`, si consideri la struttura `Employee` dichiarata precedentemente in 4.1 e si consideri `a.equals(b)`:

Relazione	Validità	Controesempio
Stesso nome (Come se fosse chiave primaria)	Valido	
Entrambi salari > 1000	Non rispetta la riflessività	a.salary = 500 a.equals(a) = false
a.salary ≥ b.salary	Non rispetta la simmetria	a.salary = 500 b.salary = 200 a.equals(b) = true b.equals(a) = false
Stesso nome oppure stesso salario	Non rispetta la transitività	x.nome = Luca x.salario = 4000 y.nome = Luca y.salario = 3500 z.nome = Marco z.salario = 3500 x.equals(y) = true y.equals(z) = true x.equals(z) = false
Un equals che ritorna sempre true	Valido	
Un equals che ritorna sempre false	Non rispetta la riflessività	x.equals(x) = false

4.2 Criteri di uguaglianza

Ipotizziamo la seguente struttura:



Distinguiamo due criteri di uguaglianza: *Criterio uniforme* e *Criterio non uniforme*.

4.2.1 Criterio uniforme

La radice della gerarchia possiede un campo identificativo ereditato da tutti, questo è il caso più semplice ed ha una implementazione soltanto in `Person` come segue:

```
1  class Person {
2    @override
3    public final boolean equals(Object other) {
4       if(!(other instanceof Person)) {
5          return false;
6       }
7       Person p = (Person) other;
8       return name.equals(p.name);
9    }
10 }
```

In questo criterio vogliamo che un `Employee` ed un `Manager` con lo stesso nome siano la stessa persona, ovvero che i confronti misti tra sottoclassi siano considerati come se fossero la stessa classe. Inoltre nella riga 3 abbiamo imposto il metodo come `final`, così facendo impediamo l'override nelle sottoclassi in quanto questo metodo dev'essere uguale per tutti.

4.2.2 Criterio non uniforme

A differenza del criterio precedente vogliamo che i casi misti come un confronto tra `Employee` ed un `Manager` siano considerati sempre diversi ed inoltre ogni sottoclasse può opzionalmente raffinare il metodo della superclasse.
Per realizzare questo ci servono controlli più precisi in `equals`, per realizzare questi non è più sufficiente `instanceof` in quanto tra `Employee` ed un `Manager` tornerebbe true.

Distinguiamo inoltre due tipi di criterio non uniforme:

- Criterio non uniforme *in senso stretto (o canonico)*: ovvero la sottoclasse raffina il metodo
- Criterio non uniforme *in senso ampio*: ovvero la sottoclasse non raffina il metodo

Per implementare il criterio non uniforme è necessario un piccolo accenno sulla *riflessione*, maggiori informazioni sono nel capitolo 4.2.3, questo ci permette di un controllo preciso di tipo.

Vogliamo anche evitare di duplicare il codice, ovvero i controlli degli attributi delle superclassi vengono controllati dalla superclasse stessa tramite un sistema di "delegazione", da tutte queste premesse concludiamo con la seguente implementazione:

```
1  class Person {
2    @override
3    public boolean equals(Object other) {
4       if(other == null) { return false; }
5
6       if(other.getClass() != getClass()) { return false; }
7
8       Person p = (Person) other;
9       return p.nome.equals(name);
10   }
11 }
12
13 class Employee {
14   @override
15   public boolean equals(Object other) {
16      if(!super.equals(other)) { return false; }
17
18      Employee e = (Employee) other;
19      return salary == e.salary;
20   }
21 }
```

Analizziamo le righe più importanti:

- Nella riga 4 facciamo il controllo che `other` non sia null, questo prima era effettuato in maniera automatica da `instanceof`, ma ora è da gestire manualmente in quanto fare il `getClass()` di un riferimento nullo lancerebbe `NullPointerException` a runtime.
- Nella riga 6 avviene il controllo di tipo, ovvero otteniamo il Class di `other` e lo confrontiamo con il Class dell'oggetto chiamante del metodo, si noti che `getClass()` è equivalente a scrivere `this.getClass()`
- Nella riga 16 deleghiamo il compito di verificare se il nome coincide alla classe `Person` richiamando `super.equals(other)`, facciamo il not del ritorno perché vogliamo che se il nome coincide, quindi ritorna true, procediamo a controllare il salario, altrimenti ritornare direttamente false.
- Nella riga 18 procediamo a fare il cast ad `Employee` in quanto se ha avuto successo il controllo precedente allora sappiamo che `other` può essere castato, i casi misti falliscono direttamente alla riga 6.

4.2.3 Accenni sulla Riflessione

Con *riflessione* intendiamo la possibilità del programma di "scoprire" le caratteristiche di un oggetto come attributi e metodi a runtime senza saperne il tipo, in alcuni casi questa funzionalità permette anche di richiamare alcuni metodi di quell'oggetto come i costruttori, grazie a questo è possibile creare oggetti identici ad esso continuando a non saperne il tipo.

Questo avviene in quanto ogni volta che viene creato un oggetto la JVM (*Java Virtual Machine*) crea anche un oggetto di tipo `Class`, gestito sempre dalla JVM, corrispondente ad esso che permette di scoprirne le caratteristiche.

In `Object` troviamo il metodo `public Class getClass()`, questo metodo restituisce l'oggetto `Class` corrispondente al tipo effettivo di questo oggetto, ad esempio se si fa `x.getClass()` si otterrà l'oggetto di tipo `Class` corrispondente al tipo effettivo di `x`.

Una volta ottenuto l'oggetto `Class` si utilizzano i vari metodi di `Class` per scoprire i suoi campi ed i suoi metodi, in alternativa è anche possibile utilizzare l'operatore `.class` di una classe che è un riferimento al suo oggetto `Class`.

4.2.4 Esempi di criteri

Si prenda la stessa struttura dichiarata in 4.2 senza il Manager, ipotizziamo i seguenti vari casi:

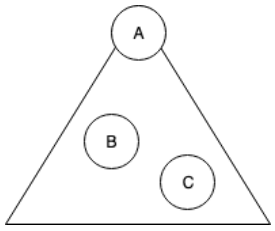
2 Person	Person / Employee	2 Employee	Validità	Scenario
Stesso nome	Mai uguali	Stesso nome	Sì	Criterio non uniforme nel senso più ampio
Stesso nome	Stesso nome	Stesso nome	Sì	Criterio uniforme
Stesso nome	Mai	Stesso nome e salario	Sì	Criterio non uniforme ma quello in senso stretto
Stesso nome	Stesso nome	Stesso nome e salario	No, non rispetta la transitività	Nessuno scenario in quanto non valido
Stesso nome	Stesso nome e salario = 0 per l'Employee	Stesso nome e salario	Sì	Scenario 3, non in programma

5 Lezione 5

La lezione 5 riprende un po' sull'equals con overriding ed overloading, classi interne e le loro proprietà per concludere con iteratori.

5.1 Ricapitolazione sull'equals

Questa sezione è solo una rapidissima ricapitolazione della lezione 4, si veda quella per tutte le informazioni più dettagliate.
Sia data la seguente gerarchia, ovvero con A superclasse, B e C sottoclassi di A:



Distinguiamo due scenari in una gerarchia di classi con uguaglianza strutturale:

1. Lo scenario uniforme che viene dichiarato nella superclasse, è final ed è indipendente dalle singole classi
2. Lo scenario non uniforme dove la superclasse ha un criterio di uguaglianza e le sottoclassi *possono* renderlo più specifico ma non è necessario, inoltre gli oggetti di tipo diverso sono **sempre** diversi.
3. Il terzo scenario, non argomento di studio, dove oggetti misti possono avere una condizione di uguaglianza.

5.2 Dynamic binding di equals

Come ogni altro metodo di Java anche `equals` è soggetto ad overloading ed overriding, però a differenza di altri è altamente consigliato di fare **sempre overriding** di `equals` in quanto farne l'overloading può avere comportamenti inaspettati del programma dove valori strutturali uguali sono considerati diversi.

Si veda il seguente esempio, considerando una classe `Employee` con attributi `String` e `int`:

```
1 Employee e1 = new Employee("Pippo", 1000);
2 Employee e2 = new Employee("Pippo", 1000);
3 Object x1 = e1;
4 Object x2 = e2;
```

Facciamo una tabella con le varie combinazioni, facendo overloading ed overriding, vogliamo che tutte le combinazioni siano true in quanto gli oggetti rispettano l'uguaglianza strutturale:

Istruzione	Overloading: equal(Employee e)	Overriding: equals(Object o)
x1.equals(x2)	False: x1 è un Object quindi prende il suo metodo equals che è di uguaglianza di identità e non strutturale	True
x1.equals(e2)	False: Stesso motivo di sopra	True
e1.equals(x2)	False: x2 è un Object quindi non coincide con le firme di Employee e sale a prendere quello di Object	True
e1.equals(e2)	True: Questo è l'unico caso in cui prende il metodo con effettivo overloading	True

5.3 Classi interne

Una classe interna è, banalmente, una classe dichiarata dentro una classe *Top-level*, in questa sezione verranno illustrate le loro proprietà e le best practices su quando usarle.

Una classe interna dispone delle seguenti tre proprietà:

1. *Restrizioni di visibilità*: Una classe interna può essere dichiarata con qualsiasi delle 4 visibilità, ovvero `default`, `private`, `protected` o `public`, rispetto ad una Top-level che può essere esclusivamente `default` o `public`, infatti uno dei motivi principali per utilizzarle è che rafforzano l'incapsulamento

2. *Privilegi di visibilità*: Apparentemente in constrasto con la proprietà precedente, le classi interne possono vedere tutti gli attributi e metodi della classe che la contiene così come la classe contenitore può vedere tutti gli attributi delle classi interne, ovvero dall'esterno le classi sono incapsulate ma tra di loro sono accessibili. In C++ queste vengono chiamate "classi amiche" e sono identificate dalla keyword `friend` che avvisa il compilatore di ignorare l'incapsulamento tra loro due.

   "Le classi tra di loro non hanno segreti"
   - M. Faella

3. *Collegamento implicito tra oggetti*: Le classe interna ha un riferimento automatico alla classe contenitrice, questa proprietà però è posseduta esclusivamente da classi interne non statiche.

Si noti che il modificatore `static` in una classe interna ha il solo scopo di negare la terza proprietà ed è importante da non confondere con quello degli attributi e dei metodi.

5.3.1 Esempio di classe interna

Sia dato il seguente codice di esempio:

```
1  class A {
2    private int n;
3    public int k;
4
5    // Proprieta' 1: la classe e' privata
6    private class B {
7       private int i;
8       private void f(A a) {
```

```
9         // Proprieta' 2: posso utilizzare n in B anche se privata
10        System.out.println(a.n);
11
12        // Proprieta' 3: siccome non c'e' un k in B allora e' implicitamente riferito
              a quello di A, si noti che le due istruzioni sono equivalenti
13        System.out.println(k);
14        System.out.println(A.this.k);
15     }
16   }
17 }
```

Se B fosse stato statico, quindi se avessimo negato la terza proprietà, nelle righe 13 e 14 avremmo errore di compilazione in quanto non esiste un valore `k` in B.

Ipotizziamo adesso di voler instanziare A e B in un Main contenuto in A, possiamo farlo così:

```
1  class A {
2    ...
3    private class B { ... }
4
5    public static void main[] {
6       A a = new A();
7       //Siccome siamo in A posso istanziare B direttamente
8       B b = a.new B();
9       //Se B fosse statico potrei istanziarlo senza riferimenti alla classe Top-level
10      //B b = new B();
11   }
12 }
```

Adesso ipotizziamo che il main sia in un altro file e proviamo ad istanziare le stesse classi precedenti:

```
1  public static void main[] {
2    A a = new A();
3    //Errore di compilazione: B e' privata ed incapsulata in A, non si puo' istanziare
         in maniera diretta
4    B b = a.new B();
5  }
```

5.4 Accenni sugli iteratori

Gli iteratori sono utilizzati nel JCF (*Java Collection Framework*) e svolgono il ruolo di recuperare elementi uno ad uno e compiere operazioni di lettura o rimozione in una collezione di oggetti, un iteratore è principalmente un interfaccia che richiede di implementare determinati metodi che saranno approfonditi nella Lezione 6.

In Java hanno la seguente implementazione semplificata:

```
1  interface Iterator {
2    boolean hasNext();
```

```
  Object next();
}
```

5.4.1 Esempio con iteratori e proprietà di classi interne

Ipotizziamo di voler implementare una lista concatenata in Java, prima di tutto partiamo con la sua interfaccia pubblica o "punto esterno", ovvero quali saranno i suoi casi d'uso: dichiarazione di essa, inserimento di un elemento in coda e scorrimento della lista.

1. Iniziamo con la dichiarazione della lista, come scelta implementativa voglio che parta vuota, quindi il caso d'uso sarà:

```
List myList = new List();
```

2. Come aggiunta voglio richiamare un eventuale metodo `add()` come segue:

```
myList.add("ciao");
```

3. Per lo scorrimento invece voglio ipotizzare un caso con un while della forma:

```
while(myList.hasNext()) {
  System.out.println(myList.item());
}
myList.reset();
```

   Qui avrò che il metodo `hasNext()` verifica se esiste e si sposta man mano, successivamente un metodo `reset()` per tornare in cima alla lista, il seguente ciclo avrebbe una complessità lineare e non prevede la gestione di indici ma non funzionerebbe con i cicli innestati.

4. Per lo scorrimento però posso anche valutare un ciclo for come segue:

```
for(int i = 0; i < myList.length(); i++) {
  System.out.println(myList.get(i);
}
```

   Il vantaggio di questa implementazione è l'accesso diretto come negli array ma presenta altri problemi rispetto la precedente, prima di tutto ha una complessità quadratica in quanto `get(i)` dovrà spostarsi ogni volta in quella posizione.

5. La soluzione migliore e più efficiente, che ci permetterebbe anche cicli innestati, è quella di utilizzare un iteratore come segue:

```
Iterator i = myList.Iterator();
while(i.hasNext()) {
  Iterator j = myList.Iterator();
  Object a = i.next();
  while(j.hasNext()) {
     Object b = j.next();
     print(a, b);
```

```
  }
}
```

In questo specifico caso se avessimo `myList = {1, 2, 3}` il suo output sarebbero tutte le possibili coppie di valori, ovvero: {(1,1), (1, 2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)}

Dopo aver stabilito l'interfaccia esterna possiamo procedere con l'interfaccia interna, ovvero la sua implementazione:

```
public class List {
  private static class Node {
     Node next;
     Object val;
     Node(Node n, Object v) { /*classico costruttore*/ }
  }
  private Node head, tail;

  public void add(Object v) {
     Node n = new Node(null, v);

     if(tail != null) {
        tail.next = n;
     }
     else {
        head = n;
     }
     tail = n;
  }

  private class ListIterator implements Iterator {
     Node current;
     ListIterator() {
        // Sto usando entrambe le proprieta' 2 e 3 qui
        current = head;
     }

     boolean hasNext() {
        return current != null;
     }

     Object next() {
        Object result = current;
        current = current.next;
        return result.val;
     }
  }

  public Iterator iterator() {
     return new ListIterator();
  }
}
```

Alcune osservazioni su questa implementazione:

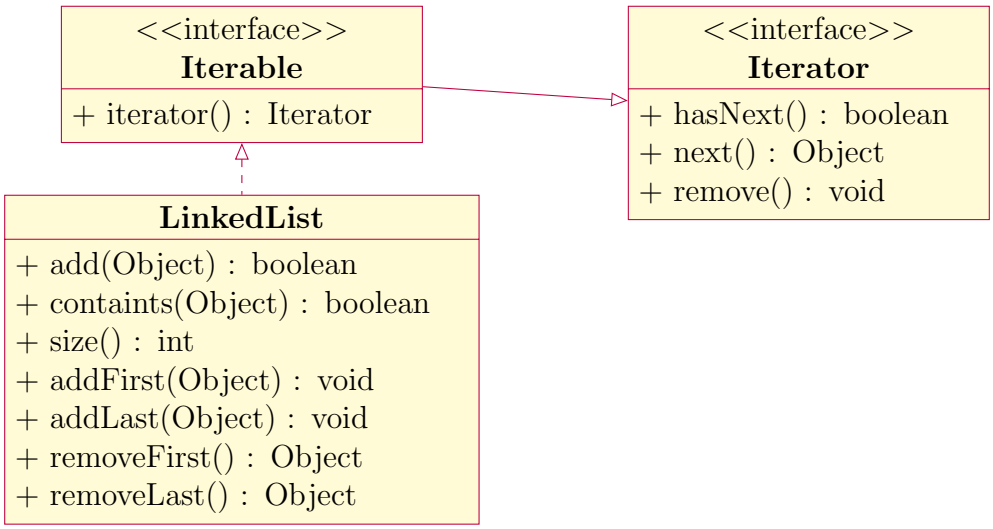
- Nella riga 2 dichiariamo una classe interna `Node` come `static` in quanto non vogliamo che ogni nodo abbia il riferimento alla sua lista
- Nelle righe 3 e 4 non dichiariamo `Node` e `val` come `private` in quanto siccome la classe è privata, anche loro lo sono
- Nella riga 7 dichiariamo un nodo testa ed uno coda per rendere costante l'eliminazione e l'inserimento in coda.
- Nella riga 21 viene creata una classe interna che implementa Iterator, come vuole l'iterator pattern, si noti che per l'implementazione consideriamo che current parta dalla testa e facciamo controlli sul nodo attuale in quanto ci evita seccature se la testa fosse null.
- Nella riga 28 che verifica se il nodo ha effettivamente un valore, si legga come un "Prossimo valore da mostrare" piuttosto che "Prossimo valore in lista"
- Nella riga 32 si ritorna il valore current e lo si sposta al successivo

6 Lezione 6

La lezione 6 mostra l'implementazione ufficiale delle Linked list e degli Iteratori nella libreria standard ufficiale, le classi locali, anonime ed infine un esercizio sui memory layout per comprendere le proprietà delle classi interne.

6.1 Linked List e Iteratori nella JDK

Questo è il primo accenno al JCF (*Java Collection Framework*), in particolare quello relativo alle Linked List e agli iteratori, è da notare però che è la versione relativa a Java 7, ovvero quella prima dell'introduzione dei *Generics*, la loro implementazione segue rappresentata in UML:



È importante notare alcuni dettagli di questo diagramma:

- Il prefisso <<interface>> può essere anche rappresentato con <<type>>
- In questo diagramma viene utilizzato l'iterator pattern
- Il metodo `add` restituisce un `boolean` ma in realtà ha sempre successo eccetto l'unico caso in cui finisca la memoria, la presenza di questo tipo di ritorno è causata per il fatto che il metodo `add()` è ereditato da un'altra interfaccia che verrà approfondita successivamente ed in alcuni casi può fallire, ma non nelle Linked List
- Il metodo `contains` utilizza `equals` per verificare l'uguaglianza
- Il metodo `remove` di Iterator è un implementazione "facoltativa", cioè deve essere necessariamente implementata oppure il compilatore da errori, però se si vuole impedire la rimozione allora la documentazione stabilisce di implementare che lanci l'eccezione unchecked `UnsupportedOperationException`

6.2 Classi locali

Una classe locale è una classe dichiarata dentro un metodo, una classe locale non può avere visibilità in quanto sono localizzate e visibili esclusivamente all'interno metodo, inoltre se il metodo è `static` allora anche la classe locale sarà `static` o viceversa, non è possibile avere casi misti.

Le classi locali possono accedere alle variabili del metodo ed ai parametri in ingresso a patto che questi siano `final` oppure *"Effectively final"* ovvero che dalla loro dichiarazione nel metodo al loro utilizzo nella classe locale non vengano mai modificate, possono anche non avere il modificatore `final` anche se è fortemente consigliato per motivi di leggibilità e chiarezza.

Si prenda il seguente codice di esempio che è completamente lecito:

```
class E {
  void f(int n) {
     double x = 42;

     //Se qui ci fosse x = 0 allora si verificherebbe un errore di compilazione
     class A {
        void g() {
           System.out.println(x); // Cattura x in quanto Effectively Final
        }
     }
     A a = new A();
  }
}
```

Questo vincolo è imposto da un elaborato meccanismo di "cattura delle variabili" che lavora in background e ogni volta che viene istanziata una classe locale copia i valori del metodo.

Si è scelto questo vincolo per impedire casi che potrebbero causare confusione al programmatore, si noti il seguente esempio:

```
void f2(int n) {
  class A { int n; }

  A a1 = new A();
  n++;
  A a2 = new A();
}
```

Qui avremmo che `a1` ha una copia del valore originale di n mentre `a2` ha il valore incrementato, questo potrebbe creare confusione e richiederebbe di studiare il meccanismo della cattura al programmatore ma i progettisti hanno scelto di evitarlo e lasciarlo in background obbligando semplicemente che esse siano "effectively final".

6.2.1 Vantaggi delle classi locali

Il vantaggio delle classi locali è quello di rafforzare uno dei principi della OOP (*Object Oriented Programming*), ovvero quello del *"Minimo privilegio"*, in questo caso se abbiamo una classe che viene utilizzata in un solo metodo allora è buona norma renderla locale.

Definizione 6.1 (Minimo Privilegio) *Se una classe può avere una visibilità minore si deve dichiarare con quella minore.*

Un altro vantaggio non indifferente è quello di poter implementare interfacce con una classe che non serve all'esterno, questo è particolarmente importante con gli iteratori, si noti il seguente esempio:

```
class E {
  Iterator f(int n) {
     // Si noti che MyIterator "cattura n"
     class MyIterator implements Iterator {
        public Object next() {
           System.out.println(n);
        }
        // Altro codice
     }
     return new MyIterator;
  }
}
```

6.3 Classi anonime

Sia Interface una generica interfaccia, una classe anonima prende la seguente forma:

```
Interface i = new Interface() {
  <body>
}
```

A primo impatto può sembrare di star istanziando un interfaccia, ma in realtà quel codice corrisponde esattamente a:

```
class _ implements Interface {
  <body>
}
```

Indicato con il trattino basso è come dire una classe senza nome, che corrisponde con il loro nome di "classe anonima", inoltre l'assenza di un nome alla classe impone anche il vincolo di non poter avere costruttori per quella classe, altro vincolo è che una classe anonima può implementare una sola interfaccia alla volta.

Una classe anonima può anche essere una classe, sia A una classe, in quel caso prende la seguente forma:

```
A a = new A(e1, e2, ..., en) {
  <body>
}

// Corrisponde esattamente a:
class X extends A {
  <body>
  A a = new X(e1, ..., en);

```

```
  public X(t1, t2, ..., tn) {
     super(t1, t2, ..., tn);
  }
}
```

Si noti che anche se rappresentato con `X` per chiarezza la classe anonima non ha un nome effettivo come per le interfacce, nonostante non abbia un nome una classe anonima ha comunque un suo oggetto Class associato ottenibile eseguendo `getClass()`.

Le classi anonime vengono utilizzate principalmente per interfacce piuttosto che per le classi, inoltre sono costrutti utilizzati principalmente in metodi, questo tipo di classi sono ancora più restrittive rispetto la classi locali ma sono meno restrittive rispetto le *lambda-espressioni* introdotte in Java 8, che sono specificate nel capitolo 10.4.

6.3.1 Esempi di classi anonime

Le classi anonime sono particolarmente importanti in Java Swing/awt o in linguaggi ad eventi, si notino i seguenti esempi rispettivamente con un interfaccia ed una classe:

1. Vogliamo creare un implementazione di ActionListener che è un interfaccia che corrisponde all'evento che si attiva alla pressione di un bottone, sarebbe inutile creare una classe per implementare quella singola funzione, qui torna utile una classe anonima:

```
JButton b = new JButton();
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
     //Codice di actionPerformed
  }
});
```

2. Ipotizziamo di avere la solita classe `Employee` e vogliamo che `Employee` speciale che abbia un titolo prima del suo nome in più e non vogliamo farne un intera sottoclasse, possiamo usare una classe anonima come segue:

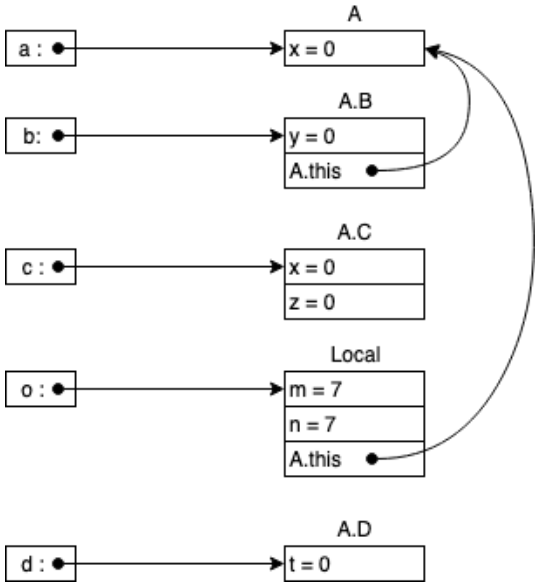
```
class Employee {
  String name;
  int salary;

  //Rendiamo title final per ricordarci che la usiamo in una classe locale
  public static Employee makeSpecialEmployee(String name, int salary, final
      String title) {
        public String toString() {
        return title + super.toString();
     };
  }
}
```

6.4 Memory layout di classi interne

Date le informazioni sulle proprietà delle classi interne ed illustrate quelle locali, è possibile visualizzarne una rappresentazione grafica dal seguente codice:

```
class A {
  private int x;

  public class B {
     private int y;
  }

  public static class D {
     private int t;
  }

  public static class C extends A {
     private int z;
  }

  pubic Object f(final int n) {
     class Local {
        private int m = n;
     }
     return new Local();
  }
}

public static void main[] {
  A a = new A();
  B b = a.new B();
  A c = new C();
  Object o = a.f(7);
  D d = new D();
}
```

Il suo memory layout avrà la seguente forma:



Alcune note sul memory layout:

- In `B`, per la terza proprietà delle classi interne, mettiamo un riferimento implicito ad `A` in quanto non è `static`.
- In `C` mettiamo un'altra `x` perché `C` è una sottoclasse di `A`, quindi eredita quel valore, inoltre a differenza di `B` abbiamo che `C` è dichiarato come `static`, quindi il suo riferimento implicito è assente.
- In `Local` abbiamo un riferimento implicito in quanto il metodo non è `static` , inoltre "cattura n" che avrà il valore in ingresso e poi lo assegna ad `m` avendo concretamente 2 variabili uguali.

7 Lezione 7

La lezione 7 mostra i Generics, il type inference, List parametrica e l'enhanted for.

7.1 Generics

È il termine che utilizza Java per indicare classi e metodi parametrici, ovvero delle classi e metodi che hanno un parametro speciale: il tipo di dato.
I Generics sono un aspetto presente in molti linguaggi di programmazione, ad esempio in C++ sono chiamati "Template".

7.1.1 Classi parametriche

Per capirne il funzionamento ipotizziamo di voler creare una classe che gestisce coppie di oggetti di qualsiasi tipo:

```
1  class Pair {
2    // Per avere coppie di ogni tipo le dichiariamo come Object
3    private Object first, second;
4    public Pair(Object a, Object b) { /*Classico costruttore*/ }
5    public Object getFirst() { return first; }
6  }
7
8  // Esempio di utilizzo
9  public static void main[] {
10   // Ipotizziamo una coppia di stringhe
11   Pair p = new Pair("a", "b");
12   String s = (String) p.getFirst();
13 }
```

Come è possibile vedere ci sono alcune cose che non vanno bene nella seguente implementazione: Il cast è pericoloso, se tra la dichiarazione e la lettura ci fosse un assegnamento di tipo errato questo causerebbe un eccezione a runtime, questo rende il codice meno solido. Però allo stesso tempo creare `Pair` come `String` porterebbe ad una perdita di potenza.

Vediamo ora la versione con i Generics:

```
1  class Pair<T> {
2    private T first, second;
3    public Pair(T a, T b) { /*Classico costruttore*/ }
4    public T getFirst() { return first; }
5  }
6
7  // Esempio di utilizzo
8  public static void main[] {
9    Pair<String> p = new Pair<String>("a", "b");
10   Pair<String> p = new Pair<>("a", "b");
11   String s = p.getFirst();
12 }
```

Analizziamo la seguente implementazione:

- Nella riga 1, vicino la dichiarazione del nome della classe, avviene la dichiarazione del *parametro formale di tipo* chiamato `T`
- `T` è visibile in tutto `Pair` escluse le zone statiche della classe (anche se in questo caso non ci sono)
- Nella riga 9 dichiariamo un instanza `Pair` che ha come parametro di tipo `String`
- Le dichiarazioni alle righe 9 e 10 sono equivalenti, alla 10 si utilizza l'operatore *diamond* <>
- Specificando `<String>` vicino a `Pair` otteniamo un aumento dei controlli da parte del compilatore
- Nell'utilizzo non abbiamo più il cast pericoloso, aumentando la robustezza del codice

Concretamente utilizziamo il Generics perché adesso le coppie dovranno essere sempre di quel tipo T, se provassi ad assegnare un tipo diverso da quello dichiarato avrei direttamente un errore a tempo di compilazione piuttosto che a runtime.
Nell'esempio precedente dichiaravamo una coppia di Stringhe `p`, di conseguenza `p` accetterà solo stringhe al suo interno.

È convenzione utilizzare singole lettere maiuscole per indicare i parametri di tipo formali, anche se è possibile scriverli in ogni modo con conseguenze più o meno pericolose, si veda nel dettaglio 7.1.4.

7.1.2 Molteplici parametri di tipo

I parametri di tipo possono essere anche molteplici, si prenda una coppia di oggetti implementata come segue:

```
1  class Pair<T, U> {
2    private T first;
3    private U second;
4    public Pair(T a, U b);
5    public T getFirst() { return first; }
6    public U getSecond() { return second; }
7  }
8
9  // Esempio di utilizzo
10 public static void main[] {
11   // I tipi di dato possono anche coincidere
12   Pair<String, String> p = new Pair<>("a", "b");
13 }
```

7.1.3 Retrocompatibilità con i Generics

L'introduzione dei Generics è avvenuta con Java 8, prima di loro esisteva già molto codice in quanto Java era popolare ed i progettisti non volevano invalidare il codice precedente, per questo motivo è possibile vedere in giro classi generiche come *"versioni grezze"*, sono **altamente sconsigliate** e sono scritte come segue:

```
// La classe Pair e' quella definita precedentemente come Pair<T, U>
Pair p = new Pair("a", "b");
String s = (String) p.getFirst()
```

Ovvero anche se `Pair` è parametrica è possibile utilizzarla come se fosse dichiarata come una coppia di `Object`.

7.1.4 Errori logici con i Generics

Si noti che un uso errato dei Generics può portare a numerosi errori logici, ad esempio una dichiarazione simile in `Pair` sarebbe illegale:

```
private static T x;
```

Questo è illegale perché ogni istanza di `Pair` può avere un `<T>` di tipo diverso e non può esistere un parametro di tipo `T` condiviso da tutte le classi.

Un altro errore pericoloso è il nome dato al parametro di tipo che potrebbe anche fare *shadowing*, si noti il seguente esempio:

```
class Person<String> {
  String name;
  static String city;
}
```

"Il seguente codice non da errori di compilazione e compila con successo, infatti vi consiglio di utilizzarlo per stupire parenti e compagni."

- M. Faella

Eseguendo questo codice avremo dei comportamenti inaspettati, ovvero avremo che il parametro di tipo `String` fa shadowing su `String` e quindi si riferisce ad un qualsiasi tipo e non un effettiva stringa però paradossalmente il campo `static` si riferisce effettivamente ad una stringa in quanto i parametri di tipo non sono applicati ai campi statici della classe.

7.1.5 Metodi parametrici

Non solo le classi ed interfacce possono essere parametrici ma anche i metodi, si veda il seguente esempio:

```
// La classe puo' anche essere non parametrica
class Test {
  // Viene dichiarato prima del valore di ritorno che puo' essere anche il tipo stesso
  public static <T> T getMiddle(T[] a) {
     int l = a.length();
     return a[l/2];
  }
}

// Utilizzo, si supponga un array di stringhe di nome x
public static void main[] {
```

```
  String s = Test.<String>getMiddle(x); // Chiamata estesa
  String s = Test.getMiddle(x); //Valido, si utilizza il Type Inference
}
```

Quando si richiama il metodo parametrico è possibile dichiarare il tipo del metodo che si vuole chiamare oppure sfruttare il *Type Inference* per lasciare dedurre al compilatore il metodo che si intende chiamare, è importante notare che questo segue le regole di assegnabilità per dedurre il tipo e può avere comportamenti anomali, si veda 7.1.6.

Un metodo parametrico può essere `static` ed allo stesso tempo usare Generics in quanto la keyword `static` ha un significato diverso per i metodi ed il tipo `<T>` è dichiarato dopo, un metodo si dice parametrico **esclusivamente** se dichiara dei metodi parametrici suoi, non se ne utilizza altri.

Invece per le visibilità si veda il seguente esempio:

```
class A<T> {
  T x;
  <U> void f(T y) { U u; } // Vede sia T che U
  static <Z> Z g() { ... } // Non vede ne' T ne' U ma solo Z
  static void h() { ... } // Non vede nessuno
}
```

7.1.6 Casi di errore con il Type Inference

Il type inference può avere comportamenti anomali, sia prenda il seguente metodo parametrico:

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

// Dichiarazioni nel main
String [] a = ...
Object[] x = a;
Object n = 7;
```

Ipotizziamo di voler far crashare la chiamata ad un metodo parametrico senza usare il type inference:

```
// Nel main
Test.<Object>fill(x, b); // Crasha in Runtime in quanto a e' una classe String
    mascherata da Object
Test.<String>fill(a, n); // Ma con il caso opposto il compilatore segnala errore
```

Come si può vedere il crash è causato da un particolare procedimento di mascheramento.

Adesso vediamo un crash con il type inference:

```
// Nel main
Test.fill(a, "ciao"); // Compila con successo, il compilatore capisce che T = String
Test.fill(a, new Integer(42)); // Compila con successo, c'e' un tipo comune ad
    entrambi, ovvero deduce che T = Object ma crasha a runtime
```

La morale è che il Type Inference permette di scrivere codice più pulito ma rende vulnerabili ad alcuni rischi e possibilità di errore, quindi è consigliato optare per la versione estesa.

7.1.7 Versione parametrica di List

La versione corretta di Iterator, tenendo conto dei Generics:

```
interface Iterator <T> {
  T next();
  ...
}

interface Iterable <T> {
  Iterator<T> iterator();
}

// Si noti che T viene subito utilizzato in Iterable dopo la dichiarazione
class LinkedList<T> implements Iterable<T> {
  public boolean add(T x) { ... }
  public T removeFirst() { ... }
  public boolean contains(Object x) { ... }
  public Iterator<T> iterator() { ... }
}
```

Si noti che la funzione `contains` accetta un `Object` in quanto non è un operazione con assegnamenti o operazioni pericolose ma al massimo ritorna `false`.

Caso d'uso:

```
LinkedList<String> myList = new LinkedList<>();
myList.add("ciao");
Iterator<String> i = myList.iterator();
while(i.hasNext()) {
  String s = myList.next();
}
```

7.2 Enhanted for (Foreach)

Tutto quello visto precedentemente dalla dichiarazione dell'iteratore al ciclo while con i Generics può essere ridotto in:

```
for(String s : myList) { ... }
```

Questo funziona con ogni classe che implementa Iterable e funziona anche sugli array, per uscire dal ciclo è necessario usare un `break` mentre per saltare alcune iterazioni un `continue`, inoltre l'iteratore e l'indice sono nascosti, questo può essere un bene se non necessari ma obbliga ad utilizzare la forma estesa se servono.

Generalizzando la forma abbiamo:

```
1 class A implements Iterable<X> { ... }
2 A a = ...;
3 for(Y y : a) { ... }
```

Affinché questo funzioni `X` deve essere un sottotipo di `Y` oppure `A` dev'essere un `X[]`.

8 Lezione 8

La lezione 8 mostra un esercizio sull'enhanced for, il JCF, le Tag interfaces ed il rapporto tra Generics ed i tipi.

8.1 Esercizio sull'enhanced for

Sia dato il seguente codice, creare una classe MyFor in modo che la prima istruzione si equivalente alla seconda:

```
1 for(Integer i : new MyFor(a, b, c));
2 for(Integer i = a; i < b; i += c);
```

Analizzando la richiesta possiamo dedurre che `MyFor` dovrà implementare Iterable di Integer oppure una sua sottoclasse, ma in questo caso non esistono sottoclassi di Integer siccome Integer è dichiarato come Final.

Procediamo con lo svolgimento:

```
1  public class MyFor implements Iterable<Integer> {
2    private int a, b, c;
3
4    public MyFor(int a, int b, int c) { /* Solito costruttore */ }
5
6    public Iterator<Integer> iterator() {
7       return new Iterator<Integer>() {
8          int i = a;
9
10         public boolean hasNext() {
11            return i < b;
12         }
13
14         public Integer next() {
15            int result = i;
16            i += c;
17            return result;
18         }
19      };
20   }
21 }
```

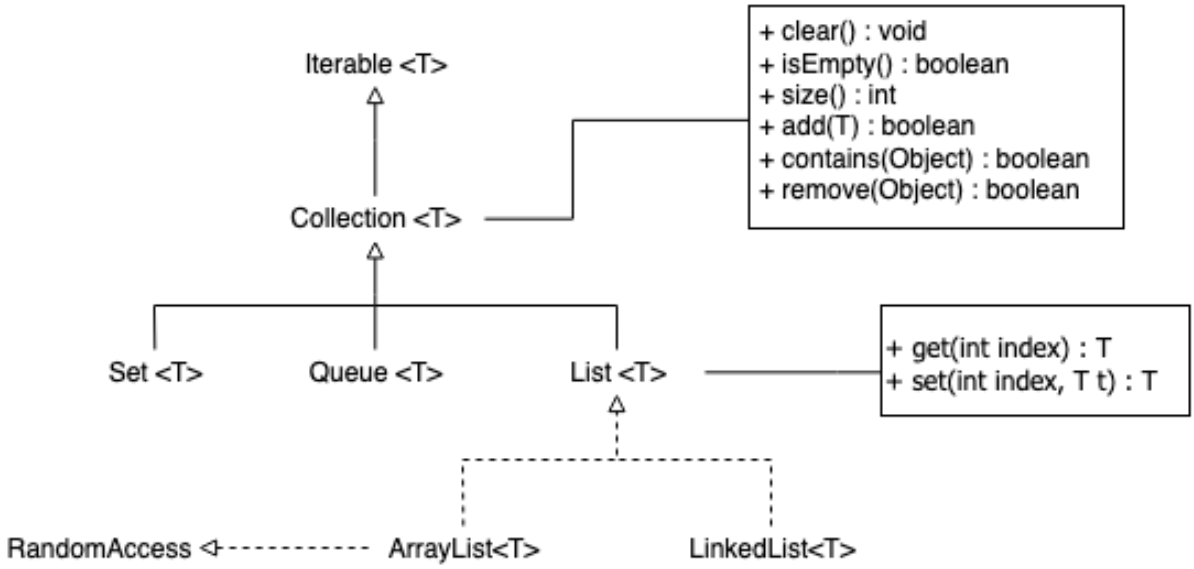
Analizziamo questa soluzione:

- Nella riga 1 dichiariamo `MyFor` non parametrica in quanto non è necessario che lo sia, bensì è una classe che implementa Iterable di Integer.
- Nella riga 2 dichiariamo `a, b,` e `c` come int piuttosto che Integer sfruttando l'auto-boxing (al capitolo 3.2.1) in quanto le operazioni aritmetiche sui Wrappers sono più costose rispetto ai tipi primitivi.

- Nella riga 6 dichiariamo il metodo che ritorna l'iteratore, si noti che l'interfaccia Iterable prevede che questo metodo non prenda parametri.
- Nella riga 7 utilizziamo di nuovo il principio del minimo privilegio e dichiariamo l'iteratore come classe anonima
- Nella riga 8 rappresentiamo l'indice, ci viene chiesto che esso inizi da a e lo inizializziamo di conseguenza.
- Nelle righe 10 e 14 implementiamo `hasNext` e `next` di Iterable, la visibilità è pubblica perché l'interfaccia prevede che questi metodi siano sempre pubblici.
- Nella riga 11 implementiamo la condizione dell'iteratore, dalla traccia ci viene richiesto che si ripete finché `i < b`.
- Nella riga 15 implementiamo il ritorno e l'incremento come richiesto nella traccia, ovvero che `i += c`.

8.2 JCF (Java Collection Framework)

La struttura del JCF è la seguente:



Da qui è possibile notare varie cose:

- Tutti i metodi più importanti delle varie collezioni sono ereditati da `Collection<T>`
- ArrayList implementa un interfaccia vuota `RandomAccess` (Si veda 8.3)
- L'interfaccia `List<T>` implementa i metodi get e set per entrambi i tipi di liste, ma con complessità diversa in base al tipo di lista, si noti che il get ritorna il valore alla posizione *i-esima* mentre il set sostiuisce il valore alla posizione *i-esima*
- Esistono due tipi di liste, le LinkedList basate tradizionalmente su nodi doppiamente puntati e l'ArrayList implementata tramite array

È importante fare una differenza fra i `Set` e le `List`, i primi possono essere visti come insiemi matematici dove l'ordine e le ripetizioni non contano, mentre i secondi possono essere visti come terne dove l'ordine e le ripetizioni contano.

Esempi:
$Set \Rightarrow \{a, b\} = \{b, a\} = \{a, a, b\}$
$List \Rightarrow (a, b) \neq (b, a) \neq (a, a, b)$

Differenziamo adesso le complessità di alcuni metodi di `Collection` per le `LinkedList` e `ArrayList`.

	LinkedList	ArrayList
add	O(1)	Dipende: Se l'array non è pieno è O(1) altrimenti se va riallocato è O(n), però secondo l'analisi di complessità ammortizzata possiamo dire che l'add è O(1)
containts	O(n): Va cercato e nel caso peggiore potrebbe non esserci	O(n)
remove	O(n): Va cercato e nel caso peggiore potrebbe non esserci	O(n): Ma possiamo dire che è anche peggio perché ogni eliminazione richiede uno shift dell'array
get/set	O(n)	O(1)
addFirst/addLast	O(1)	Non Esiste

In generale l'ArrayList è preferibile rispetto ad una LinkedList in quanto un'ArrayList, dal punto di vista della memoria, è 4 volte più leggera rispetto ad una LinkedList con gli stessi elementi.

Il motivo per cui si potrebbe scegliere una LinkedList rispetto ad un ArrayList è per l'eliminazione più rapida ed i metodi addFirst e addLast che permettono di aggiungere rispettivamente in testa e coda con tempo costante.

8.3 Tag interfaces (Interfacce vuote)

Sono interfacce puramente "contrattuali", servono a capire caratteristiche di una classe senza conoscerne il contenuto, ad esempio l'interfaccia `RandomAccess` di ArrayList ha lo scopo di far capire che ArrayList implementa i metodi `get()` e `set()` in tempo costante.
Questo può essere utilizzato in runtime per verificare se una lista implementa `RandomAccess` per utilizzare i get/set più efficienti rispetto a quelli in complessità lineare.

8.4 Generics e rapporto di tipo

Date le regole definite precedentemente in 2.4.2, stabiliamo che `A<S>` non è sottotipo di `A<T>`, questo è dovuto al fatto che per i Generics è stata privilegiata la robustezza e quindi gli assegnamenti impongono l'uguaglianza.

Si veda il seguente esempio con gli Array ed il rapporto di tipo:

```
1 String[] s = {"a", "b"};
2 Object[] x = s;
3 x[0] = new Object();    // Compila ma lancia "ArrayStoreException" a runtime
```

Ed il seguente con tipi parametrici:

```
1 ArrayList<String> s = new ArrayList<>();
2 s.add(...);
```

```
ArrayList<Object> x = s;  // Non compila
x.add(new Object());
```

8.4.1 Parametri di tipo con limite superiore

Ipotizziamo di voler creare una funzione che sommi i salari di un ipotetica classe Employee (con rispettiva sottoclasse Manager):

```
int totalSalary(List<Employee> myList) {
  ...
  for(Employee e: myList) {
     tot += e.getSalary();
  }
}
```

Questo metodo funzionerà esclusivamente con liste di Employee anche se concettualmente sarebbe identico ad una funzione di somma per una lista di Manager, per fare questo si utilizza un nuovo costrutto: Parametro di tipo con limite superiore.

Ovvero limitiamo il parametro di tipo, preso un parametro T non potrà essere qualsiasi tipo ma dovrà essere un sottotipo specificato, si prenda l'esempio precedente ma che ora vale sia per Employee che Manager:

```
<T extends Employee> int totalSalary(List<T> myList) {
  ...
  // Siccome T e' sconosciuto in compilazione, ma sappiamo che T sara' almeno
     Employee e' possibile usare il suo scorrimento
  for(Employee e: myList) {
     tot += e.getSalary();
  }
}
```

Un parametro di tipo con limite superiore può avere più limiti superiori separati da un &, però solo il primo può essere una classe mentre tutti gli altri devono essere interfacce, ad esempio:

```
<T extends Employee & RandomAccess>
```

Ovviamente anche i parametri di tipo multipli possono avere limiti superiori:

```
<S, T extends S>
```

Dalla dichiarazione di un parametro di tipo può subito seguire il suo utilizzo:

```
<S extends Iterable<S>>
```

9 Lezione 9

La lezione 9 mostra una nota sugli iteratori e si concentra principalmente sul confronto ordinale.

9.1 Nota sugli iteratori

Non è possibile fare modifiche ad una lista mentre essa sta iterando eccetto per la rimozione dell'iteratore, una modifica durante lo scorrimento invalida all'istante l'iteratore causando il lancio dell'eccezione `ConcurrentModificationException`, si prenda il seguente esempio:

```
List<Integer> integers = newArrayList(1, 2, 3);
for (Integer integer : integers) {
  integer.remove(1);
}
```

Non è legale in quanto si utilizza il remove degli integers ed invalida l'ireratore, mentre il seguente codice è corretto:

```
for (Iterator<Integer> iterator = integers.iterator(); iterator.hasNext();) {
  Integer integer = iterator.next();
  if(integer == 2) {
     iterator.remove();
  }
}
```

"Le modifiche all'iteratore lo rompono"
- M. Faella

9.2 Confronto ordinale

Con confronto ordinale intendiamo una relazione d'ordine tra oggetti, un po' come equals, in questo caso come stabilire se una classe è minore di un'altra , si prenda ad esempio di voler decidere se un certo Employee è minore di un altro Employee, per questo si utilizza la seguente interfaccia:

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

Questa interfaccia va implementata in modo che torni un valore negativo se `this` è più piccolo, un valore positivo se `this` è più grande e 0 se identici.

Si ipotizzi di voler ordinare Employee alfabeticamente per nome, si noti che implementiamo Comparable tra Employees perché vogliamo confrontare Employees, ma è facile cadere in errore mettendo String:

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

```

```
  @override
  public int compareTo(Employee other) {
     // Delego il confronto delle stringhe alla classe String
     return name.compareTo(other.name);
  }
}
```

Le classi che implementano `Comparable` si dicono *classi dotate di ordinamento naturale*, diremo che sia String che Employee sono dotate di ordinamento naturale.

Adesso ipotizziamo di voler aggiungere un altro metodo di ordinamento, ad esempio un Employee può essere ordinato anche per salario, questo è un limite di `Comparable` in quanto non è possibile che un interfaccia implementi lo stesso metodo in più modi.

Per fare questo si utilizza l'interfaccia `Comparator`:

```
interface Comparator<T> {
  int compare(T a, T b);
}
```

`compareTo` confronta `this` con `other` mentre `compare` confronta 2 oggetti e restituisce un intero con le stesse caratteristiche precedenti, ora aggiungiamo concretamente il confronto per i salari in Employee:

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

  public static final Comparator<Employee> ComparatorBySalary = new Comparator<>() {
     public int compare(Employee a, Employee b) {
        return a.salary - b.salary;
     }
  };
}
```

Utilizziamo come sempre il principio del minimo privilegio dichiarando `Comparator` come classe anonima, la dichiariamo final in modo che non sia modificabile e static perché sarà lo stesso metodo di confronto per tutti gli Employee, infine per il nome:

> *"Faccio come vuole la documentazione di Java, ovvero scrivo un nome lungo un chilometro"*
>
> *- M. Faella*

Possiamo vedere che la classe String è implementata in modo simile:

```
// Il confronto naturale e' case sensitive
class String implements Comparable<String> {
  // Creiamo un confronto case insensitive
  public static final Comparator<String> CASE_INSENSITIVE_ORDER = ...
}
```

Concludiamo quindi che è buona pratica implementare `Comparable` per avere il criterio naturale mentre `Comparator` per tutti quelli accessori.

Un esempio pratico di `Comparator` è presente nella classe `Collections`, da non confondere con la classe `Collection` senza la s, questa classe contiene algoritmi utili per le collezioni come gli ordinamenti, dei quali abbiamo due versioni:

```
// Nella classe Collections
public static <T> void sort(List<T> list, Comparator<T> comp);
public static <T> void sort(List<T> list); //Utilizza l'ordinamento naturale,
    funziona solo se gli oggetti della lista implementano Comparable
```

Anche in C esiste una versione simile di sorting, ed è il seguente:

```
void qsort(void* base, size_t nmemb, size_t size, int (*compar)(constr void*, const
    void*));
```

Questo metodo è diviso in due "pezzi": L'array che è formato dal primo parametro che è l'indirizzo base dell'array che può essere di ogni tipo, nmemb che è il numero di elementi dell'array, size che è la dimensione di un singolo elemento e dal compar che è l'analogo del Comparator.

9.3 Proprietà degli ordinamenti

Gli ordinamenti, come equals, devono rispettare le seguenti proprietà $\forall x, y, z$

1. $x.compareTo(x) = 0$

2.1. $x.compareTo(y) < 0 \iff y.compareTo(x) > 0$

2.2. $x.compareTo(y) = 0 \iff y.compareTo(x) = 0$

3. $(x.compareTo(y) < 0) \wedge (y.compareTo(z) < 0) \Rightarrow x.compareTo(z) < 0$

L'ultima proprietà è valida sia per (> 0) che $(= 0)$ come corollario delle proprietà 1 e 2.

9.3.1 Esercizio sugli ordinamenti

Quali delle seguenti sono specifiche valide tra due oggetti di tipo `String` dato il seguente criterio di confronto: `compare(String str1, String str2)`:

1. Un criterio che ritorna -1 se `str1` contiene caratteri non alfanumerici e `str2` non ne contiene, +1 se `str2` contiene caratteri alfanumerici e `str1` non ne contiene, 0 altrimenti.
   Risposta: È vacuamente transitiva perché la premessa non si verifica mai, siccome un implicazione che parte da falsa è vera, è una specifica valida.

2. Un criterio che ritorna -1 se `str1` è lunga esattamente la metà di `str2`, +1 se `str1` è lunga esattamente il doppio di `str2`, 0 altrimenti.
   Risposta: Non rispetta la transitività perché se $x.compareTo(y) < 0$ e $y.compareTo(z) < 0$ non implica che $x.compareTo(z) < 0$ in quanto z sarebbe il quadruplo di x.

10 Lezione 10

La lezione 10 mostra la programmazione funzionale, il lambda calcolo, le relative lambda espressioni ed i method references.

10.1 Functional Programming (Programmazione funzionale)

10.1.1 Lambda-calculus

Il lambda calcolo è un formalismo logico introdotto dal matematico *Alonzo Church* nel 1935, esso può essere anche considerato come il primo "Linguaggio di programmazione", esso è puramente algebrico ed è un modello di computazione universale che può essere usato per simulare anche macchine di Turing.

Si prenda ad esempio la seguente funzione: $f(x, y) := x + y$
Per il lambda calcolo i parametri di una funzione possono essere funzioni a loro volta, quindi possiamo dire che $\lambda x, \lambda y, (x + y)$

Il principio fondamentale del lambda calcolo è che tutto sia una funzione matematica, anche l'intero programma, per questo valgono le seguenti proprietà:

- Dato un determinato input, si ottiene sempre lo stesso output
- Tutte le variabili sono *stateless*, ovvero la valutazione di un esperessione non ne altera il loro valore, questa proprietà è anche detta *assenza di effetti collaterali*
- Non esistono cicli, assegnamenti e variabili, è fortemente basato sulla ricorsione

Dal lambda calcolo sono nati i **Linguaggi funzionali**, tra i più importanti abbiamo LISP (1958), MetaLanguage (1973) ed Haskell (1990).

I linguaggi funzionali portano notevoli vantaggi per la parallelizzazione, si veda il seguente esempio:
`f(g(a), h(b), g(c))`
Siccome tutte le funzioni sono prive di effetti collaterali possono essere valutate in qualsiasi ordine, potrei eseguire prima g ed h e poi fare f oppure eseguirle parallelamente, per l'assenza di effetti collaterali non potranno nascere deadlock o race condition.

10.2 Interfacce in Java 8

Prima di Java 8 le interfacce potevano avere solo dichiarazioni di metodi al loro interno ed ogni metodo era `public static final`, da Java 8 possono avere anche *metodi statici* e *metodi default*, questo permette di avere interfacce con metodi definiti e richiamabili altrove senza doverli implementare in una classe.

Ad esempio quella che prima era la classe `Math` in Java 7 è successivamente diventata una semplice interfaccia in Java 8 siccome non aveva attributi ma solo metodi.

10.2.1 Comparator post Java 8

L'interfaccia Comparator in Java 7 aveva un solo metodo da implementare, ovvero il metodo `compare`, con Java 8 ha subito un estensiva modifica ricevendo una dozzina di metodi tra cui equals.

Sempre per motivi di retrocompatibilità, in quanto il vecchio codice avrebbe imposto a tutti gli sviluppatori di implementare questi nuovi metodi, sono state aggiunte due nuove funzionalità:

1. **Metodi default** con un implementazione di default nelle interfacce, ovvero metodi concreti ma con la caratteristica che una classe che implementa l'interfaccia non deve necessariamente farne l'overriding ma opzionalmente può farlo.
2. **Metodi statici** con un corpo nelle interfacce, come i metodi statici solo che viene impedito l'overriding nella classe che le implementa.

Queste aggiunte hanno reso le interfacce più vicine alle classi astratte, ma la differenza tra queste due è che un interfaccia può avere meno visibilità rispetto ad una classe astratta (quest'ultima può essere solo `Public`) ed un interfaccia è *stateless* in quanto non può avere attributi.

Per questo diremo che le interfacce portano *"comportamenti"* ma non *"dati"*.

10.2.2 Approfondimenti su Comparator con Java 8

Date le precedenti premesse, analizziamo con un esempio la classe Comparator:

```
public class InterfacesTests {
  public static void main(String[] args) {
     Employee mike = new Employee("Mike", 2000);
     Employee louise = new Employee("Louise", 2000);

     // Dichiarazione del comparatore che ordina per nome
     Comparator<Employee> byName = new Comparator<Employee>() {
        public int compare(Employee a, Employee b) {
           return a.getName().compareTo(b.getName());
        }
     }

     System.out.println("By name:" + byName.compare(mike, louise));
     // Stampa "By name: 1" in quanto mike > louise

     //  Compare con null, siccome lancia eccezioni usiamo il try-catch
     try {
        System.out.println(byName.compare(mike, null));
     } catch(NullPointerException e) {
        System.out.println(e); //Stampa l'eccezione
     }
  }
}
```

Come si può vedere se invece di `louise` venisse passato `null` si verificherebbe il lancio dell'eccezione `NullPointerException`, proprio per questo motivo con Java 8 è stato introdotto il metodo statico `nullsLast(Comparator)` che prende un comparatore e ne restituisce una versione che fa lo stesso confronto ma supporta anche null come argomento, un valore null viene considerato piu' grande, si veda il seguente esempio per il suo utilizzo:

```
Comparator<Employee> byNameThenNull = Comparator.nullsLast(byName);
```

```
System.out.println(byNameThenNull.compare(mike, louise)); // Stampa 1 (mike > louise)
System.out.println(byNameThenNull.compare(mike, null)); // Stampa -1 (null > mike)
```

L'opposto di questo metodo, ovvero quello che considera null come più piccolo, è il metodo `nullsFirst(Comparator)`.

Un altro metodo introdotto è `reversed()` che si esegue su un comparatore e ritorna il comparatore che esegue ordinamento inverso rispetto al precedente, si veda il seguente esempio:

```
Comparator<Employee> nullThenByDecreasingName = byNameThenNull.reversed();
System.out.println(nullThenByDecreasingName.compare(mike, louise)); // Stampa -1
System.out.println(nullThenByDecreasingName.compare(mike, null)); // Stampa 1
```

10.3 Functional Interfaces - FIs (Interfacce funzionali)

Dai capitoli precedenti arriviamo al concetto di interfaccia funzionale, ovvero una qualsiasi interfaccia che abbia un singolo metodo astratto (Ma può averne anche di default o statici), alcuni esempi sono: `Runnable`, `Iterable`, `Comparator` e `Comparable`.

Un interfaccia funzionale può essere anche *"pura"*, ovvero se l'interfaccia è stateless, sono quelle interfacce la cui implementazione non prevede attributi, delle precedenti solo `Comparator` è pura, ovviamente questo non è controllato dal compilatore ma il suo contratto di implementazione (I contratti saranno approfonditi nella lezione 11) prevede che sia una classe senza attributi.
`Comparable` invece prevede attributi in quanto sono quelli che vengono confrontati concretamente.

Le interfacce funzionali pure sono quelle che richiamano il paradigma di programmazione funzionale, ovvero che possono essere considerate funzioni matematiche e che con determinati input restituiscono sempre lo stesso output, queste interfacce giocano un ruolo essenziale con gli stream. Per questo tipo di interfacce è possibile utilizzare l'annotazione `@FunctionalInterface`, questo dice al compilatore di verificare che l'interfaccia abbia un singolo metodo astratto.

Le API di Java 9 prevedono più di 40 interfacce funzionali pure, si prendano queste due per gli esempi successivi:

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
}

@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

10.4 Lambda Expressions

Le lambda espressioni si presentano come un modo compatto per implementare interfacce funzionali, queste sono un abbreviazione ancora più succinta di una classe anonima ed utilizzano

l'operatore freccia '->' che finora non era mai stato utilizzato in Java.

Prima della freccia abbiamo una serie di dichiarazioni e dopo la freccia il corpo del metodo, si veda il seguente esempio di una comparatore di stringhe basato sulla loro lunghezza:

```
Comparator<String> byLength = (String a, String b) -> {
  return Integer.compare(a.length(), b.length());
};
```

Precedentemente abbiamo visto che per confrontare gli interi era possibile fare una sottrazione tra i due, ma il metodo più robusto per farlo è utilizzare il comparatore tra interi della classe Integer in quanto considera tutti i casi speciali che potrebbero creare problemi.

Si noti che non abbiamo dovuto scrivere `new Comparator` né il nome della funzione che sto implementando, tutto questo viene fatto in automatico perché Comparator è un Interfaccia funzionale con un singolo metodo astratto.

Tutti i metodi visti nel capitolo 10.2.2 possono essere implementati come lambda espressioni, seguono le varie implementazioni che mostrano anche come possono essere ulteriormente semplificate le lambda espressioni:

```
public class LambdaTests {
  public static void main(String[] args) {

     // Comparatore visto precedentemente con classe anonima
     Comparator<Employee> byName = new Comparator<Employee>() {
        public int compare(Employee a, Employee b) {
           return a.getName().compareTo(b.getName());
        }
     }

     // Comparatore con lambda espressione
     Comparator<Employee> byNameLambda1 = (Employee a, Employee b) -> {
        return a.getName().compareTo(b.getName());
     };

     // E' possibile togliere il tipo dei parametri ed usare la Type Inference
     Comparator<Employee> byNameLambda2 = (a, b) -> {
        return a.getName().compareTo(b.getName());
     };

     // Siccome e' una sola istruzione e' possibile togliere il return e le graffe
     Comparator<Employee> byNameLambda3 = (a, b) ->
         a.getName().compareTo(b.getName());
  }
}
```

Quindi una lambda espressione può essere vista come:
`<Parameters> -> <Body>`

- `Parameters` può avere nessuna o più espressioni, esempi:
  - (int a, int b)
  - (a, b)
  - (a)
  - ()
  - a
- `Body` può essere un blocco di istruzioni racchiuse tra parentesi graffe o una singola espressione, esempi:
  - block
  - expr

Come visto nell'esempio precedente le lambda espressioni permettono molte libertà riguardo l'emettere informazioni di tipo come il tipo di parametro, quale metodo si sta implementando, il ritorno e così via, tutto questo viene riempito dalla Type Inference.

10.4.1 Type inference e lambda espressioni

Si noti però che la Type Inference funziona con le lambda espressioni esclusivamente se il contesto contiene sufficienti informazioni per identificare i dati, casi da cui la Type inference può dedurre il tipo sono:

- RHS - Right hand side, ovvero il compilatore può capire il tipo in base al lato sinistro dell'assegnazione:
  `Consumer<String> = lambda`
- Parametro di un metodo o costruttore, il compilatore va a vedere cosa il metodo si aspetta da quell'espressione:
  `new Thread(lambda)`
- Valore di ritorno:
  `return lambda`
- Valore di un cast:
  `(Consumer<String>) lambda`

Se il compilatore non identifica il valore allora segnala errori di compilazione, si vedano i seguenti esempi:

```
1  // Sintassi standard
2  Consumer<String> c1 = msg -> System.out.println(msg.length());
3
4  // Errore di compilazione: not enough info
5  Object x1 = msg -> System.out.println(msg.length());
6
7  // Errore di compilazione: not enough info
8  Object x2 = (String msg) -> System.out.println(msg.length());
9
10 // OK: Il cast rende chiaro il tipo
11 Object x3 = (Consumer<String>) ((String msg) -> System.out.println(msg.length()));
```

10.4.2 Cattura di valori delle Lambda espressioni

Valgono le stesse regole delle classi locali ed anonime, ovvero le lambda espressioni possono accedere a campi statici della classe in cui si trovano e campi instanza del metodo in cui si trovano tramite un riferimento all'oggetto che l'ha creata ed ai campi *Effectively final* tramite la loro copia.

10.4.3 Lambda espressioni vs Classi anonime

Le lambda espressioni hanno 3 vantaggi:

- Sintassi più succinta
- Non creano addizionali file .class
- Non tutte le occorrenze di una lambda espressione creano un nuovo oggetto

Per l'ultimo punto se possibile non viene creato un nuovo oggetto ma si utilizza quello precedente, questo è relativo al corpo della lambda espressione, se essa cattura valori ne viene creata una nuova, se invece è pura (usa solo i suoi parametri) viene riciclata.

Dall'altro campo le lambda espressioni hanno 2 svantaggi:

- Funzionano solo per interfacce funzionali
- Una lambda espressione non può avere campi a differenza di una classe anonima

Seguono adesso vari esempi di riciclo di lambda espressioni:

```
1  public class lambdaImplementations {
2    public static void main(Strings[] args) {
3       // Classe anonima, crea 5 classi differenti
4       for(int i = 0; i < 5; i++) {
5          Consumer<String> myPrinter1 = new Consumer<String>() {
6             @Override
7             public void accept(String msg) {
8                System.out.println("Consuming " + msg);
9             }
10         };
11         // Il toString su una classe serve a stamparne l'indirizzo
12         myPrinter1.accept(myPrinter1.toString());
13      }
14
15      // Lambda espressione che non cattura valori, viene riciclata
16      for(int i = 0; i < 5; i++) {
17         Consumer<String> myPrinter2 = msg -> System.out.println("Consuming " + msg);
18         myPrinter2.accept(myPrinter2.toString());
19      }
20
21      // Lambda espressione che cattura valori costanti, viene riciclata
22      final int secret = 42;
23      for(int i = 0; i < 5; i++) {
24         Consumer<String> myPrinter3 = msg -> System.out.println("Consuming " + msg +
               ", " + secret);
25         myPrinter3.accept(myPrinter3.toString());
```

```
    }
  }

  private int id = 1;
  public void foo() {
    // Lambda espressione che cattura valori this, non viene riciclata!
    for(int i = 0; i < 5; i++) {
      Consumer<String> myPrinter4 = msg -> System.out.println("Consuming " + msg +
          ", " + id);
      myPrinter4.accept(myPrinter4.toString());
    }
  }
```

10.5 Method references

Sono una sorta di "puntatori a funzione" del C/C++, prima di Java 8 si usava la classe `Method`, per essi si utilizza l'operatore `::` e si utilizzano come segue:

- Static method — `Employee::getMaxSalary`
- Instance method, unspecified instance — `Employee::getSalary`
- Instance method, specified instance — `mike::getSalary`
- Constructor — `Employee::new`

I method references non hanno un tipo intrinseco, anche in questo caso sono soggetti al Type Inference, quindi come per le lambda expressions se non ci sono sufficienti informazioni il compilatore segnala errori.

I method references possono essere utilizzati esclusivamente in contesti che si aspettano una lambda espressione o un interfaccia funzionale, si veda 10.4.1 per i contesti legali dove è possibile utilizzarli, essi possono essere utilizzati in maniera quasi equivalente negli stessi contesti.

Si noti il seguente esempio dove classe anonima, lambda espressione e method reference sono totalmente equivalenti:

```
Employee frank = new Employee("Frank", 3000);
Integer i = frank.getSalary();

// Method reference
Supplier<Integer> s2 = frank::getSalary;

// Lambda expression
Supplier<Integer> s3 = () -> frank.getSalary();

// Anonymous Class
Supplier<Integer> s4 = new Supplier<>() {
  public Integer get() {
    return frank.getSalary();
  }
}
```

Si noti la funzione `Function` che può essere implementata tramite classe anonima oppure method reference:

```
@FunctionalInterface
interface Function<S, T> {
  T apply(S s);
}

// Method reference - Instance method (instance not specified)
Function<Employee, Integer> f1 = Employee::getSalary;

// Anonymous class, totalmente equivalente
Function<Employee, Integer> f2 = new Function<>() {
  public Integer apply(Employee e) {
     return e.getSalary();
  }
}

// Come viene richiamata
Integer frankSalary = f1.apply(frank);
Integer frankSalary = f2.apply(frank);
```

Infine un esempio di method reference per creare un comparatore basato su un solo campo, in questo caso l'ordinamento per nome di un dipendente:

```
Comparator<Employee> byName = Comparator.comparing(Employee::getName);
```

Questo funziona in quanto il metodo ritorna una stringa e siccome le stringhe sono dotate di criterio naturale ritorna il comparatore delle stringhe.

11 Lezione 11

La lezione 11 mostra il Design by contract, il principio di sostituzione di Liskov e le relative regole dell'overriding.

11.1 Design by Contract (Programmazione tramite contratti)

È una metodologia di sviluppo software, l'idea è quella di applicare il concetto informale di contratto in programmazione, ovvero un accordo tra due parti in cui le parti si assumono degli obblighi in cambio di benefici.

Applicato ad un metodo di una classe un contratto specifica quale compito il metodo promette di svolgere e quali sono le **pre-condizioni** e **post-condizioni** richieste.

- Con **Pre-Condizione** indichiamo i valori passati al metodo come argomenti e lo stato dell'oggetto su cui viene invocato il metodo, possiamo dire che sono tutte quelle proprietà che il chiamante deve assicurare prima di invocare il metodo e sono *le responsabilità del chiamante.*
- Con **Post-Condizione** indichiamo i compiti che assolve il metodo e sono il valore di ritorno, il nuovo stato dell'oggetto ed eventuali effetti collaterali, possiamo dire che sono tutte quelle proprietà che il metodo assicura che saranno realizzate alla fine della sua esecuzione e sono *le responsabilità del metodo.*

Nella post-condizione con "effetti collaterali" intendiamo l'eventualità di stampare messaggi, scrivere su file o terminare il programma, una parte della post-condizione è la **penale**, ovvero come reagisce il metodo se la pre-condizione non è rispettata, in Java la penale è tipicamente il lancio di un eccezione.

Possiamo vedere tutto questo da due punti di vista:

- Dal punto di vista del client: Il compito svolto è un beneficio e le pre-condizioni sono obblighi
- Dal punto di vista del metodo: Il compito da svolgere è un obbligo, le pre-condizioni sono un beneficio che agevolano o consentono il compito

È importante notare che la pre-condizione deve essere **verificabile** da parte del client, ad esempio un metodo che si interfaccia con una stampante non può prevedere come pre-condizione che la stampante abbia carta a sufficienza, perché questa condizione non può essere verificabile da parte del client.

Siccome esiste un contratto tra chiamante e metodo, se a runtime non viene rispettata la pre-condizione è colpa del chiamante mentre se è violata la post-condizione è colpa del metodo, questo è il punto di forza del pattern, aiutare a *delegare responsabilità ad altri.*

In qualsiasi linguaggio di programmazione, **solo una parte del contratto può essere espressa nel linguaggio stesso**, mentre il resto sarà indicato in linguaggio naturale, ad esempio nei commenti relativi al metodo. Ovviamente il programmatore dovrebbe impegnarsi ad esprimerlo tramite codice in modo che il compilatore si occupi di verificare che sia rispettato, alcuni esempi di contratti da codice sono il numero di parametri formali di un metodo (pre-condizione) ed il tipo di ritorno (post-condizione).

Alcuni linguaggi, come *Eiffel*, o strumenti di analisi statica come *ESC/Java*, permettono di specificare formalmente una parte considerevole del contratto di un metodo.

Si veda il contratto e la clausola "throws" in Java, ambiguamente può sembrare che indichi il tipo di eccezione che viene sollevata in caso di violazione delle pre-condizioni ma questo non è corretto, in Java la clausola è consigliata solo per le eccezioni verificate quindi non dice niente riguardo pre-condizioni con penali non verificate.

Un contratto si dice **ben definito** se la pre-condizione contiene tutte le proprietà che servono al metodo per svolgere il suo compito, ovvero per realizzare la sua post-condizione.

Ad esempio si considerino i seguenti contratti:

1. Un metodo `max` che accetta due oggetti e ne restituisce il più grande tra i due.
   - Il contratto **non** è ben definito perché non è chiara la relazione d'ordine
   - Una versione corretta è che il metodo accetti due oggetti di una classe dotata di ordinamento naturale e ne ritorni il più grande
2. Un metodo `reverse` che accetta una collezione e ne inverte l'ordine.
   - Il contratto **non** è ben definito perché non tutte le collezioni sono ordinate, se la collezione fosse un HashSet non avrebbe senso
   - Una versione corretta è che il metodo accetta una lista e ne inverte l'ordine dei suoi elementi

11.1.1 Contratto di iterator

Presentiamo ora il contratto dell'interfaccia Iterator, un contratto per un interfaccia è particolarmente importante in quanto rappresenta la sua vera *raison d'etre* (il suo motivo).

`boolean hasNext()`

- Pre-condizione: nessuna, tutte le invocazioni sono lecite
- Post-condizione:
  - 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

`Object next()`

- Pre-condizione: ci sono ancora elementi su cui iterare (cioè un evocazione di `hasNext` restituirebbe true)
- Post-condizione:
  - Restituisce il prossimo oggetto della collezione
  - Fa avanzare l'iteratore all'oggetto successivo, se esiste
- Trattamento degli errori:
  - Solleva `NoSuchElementException` (non verificata) se la pre-condizione è violata

`void remove()`

- Pre-condizione:
  - Prima di questa invocazione a remove, è stato invocato next, rispettando la sua pre-condizione
  - Dall'ultima invocazione a next non è già stato chiamato remove, ovvero non è possibile evocare più remove dopo una singola next
- Post-condizione:
  - Rimuove dalla collezione l'oggetto restituito dall'ultima chiamata a next
  - Non modifica lo stato dell'iteratore (cioè non ha influenza su quela sarà il prossimo oggetto ad essere restituito da next)
- Trattamento degli errori:
  - Solleva `IllegalStateException` (non verificata) se la pre-condizione è violata

Inoltre questo metodo è indicato come opzionale nella documentazione, se non si vuole implementare è possibile lasciare l'implementazione di default che semplicemente lancia l'eccezione `UnsupportedOperationException` (non verificata)

11.1.2 Esempio di violazione di contratti

Sia dato il seguente esercizio della traccia d'esame del 21/04/2008 n.4, individuare l'output del programma e dire se la classe rispetta il contratto di Iterator, in caso contrario giustificare la risposta.

```
1  public class CrazyIterator implements Iterator {
2    private int n = 0;
3    public Object next() {
4       int j;
5       while (true) {
6          for (j=2; j<=n/2; j++)
7             if (n % j == 0) break;
8          if (j > n/2) break;
9          n++;
10      }
11      return new Integer(n);
12   }
13
14   public boolean hasNext() {
15      n++;
16      return true;
17   }
18
19   public void remove() {
20      throw new RuntimeException();
21   }
22
```

```
23    public static void main(String[] args) {
24       Iterator i = new CrazyIterator();
25       while (i.hasNext() && (Integer)i.next()<10) {
26          System.out.println(i.next());
27       }
28    }
29 }
```

L'output del programma sono i primi numeri primi minori di 10, mentre le violazioni del contratto sono le seguenti:

- `hasNext()` fa avanzare l'iteratore, questa è una violazione in quanto hasNext non dovrebbe modificarlo
- `remove()` lancia l'eccezione errata

11.1.3 Contratti ed overriding

Così come è possibile l'overriding dei metodi, è possibile "l'overriding" di contratti, questa cosa però non è possibile farla in un modo arbitrario, un contratto ridefinito in una sottoclasse deve assicurare il principio di sostituibilità.

Definizione 11.1 (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 ad ingegneria del software ed è il principio di sostituzione di Liskov (enunciato da Barbara Liskov nel 1987).

Definizione 11.2 (Principio di sostituzione di Liskov) *Se un client usa una classe A, deve poter usare allo stesso modo le sottoclassi di A, anche senza conoscerle.*

Questo è un principio cardine del polimorfismo, facendo un rapido recap sul polimorfismo:
Se io ad esempio ho `Employee sal = new Manager()`, mi aspetto di poter chiamare i metodi previsti da Employee anche se il suo tipo dinamico è un Manager, mi aspetto che se richiamo getSalary continui a ritornarmi il salario, ovvero che non si verifichi che getSalary in Employee torna il salario ma in Manager l'età.

Quindi questo è il criterio di un corretto overriding di un contratto, ovvero una sottoclasse può rendere più specifico un contratto ma senza stravolgerlo, o meglio che ogni ridefinizione del contratto dovrebbe offrire ai client almeno **gli stessi benefici**, richiedendo **al più gli stessi obblighi**, in altre parole un overriding può rafforzare la post-condizione (garantire di più) e indebolire la pre-condizione (richiedere di meno).

Tale condizione prende il nome di *regola contro-variante*, perché la pre-condizione può variare in modo opposto alla post-condizione.

Vediamo la regola contro-variante in Java, sia data la seguente generica intestazione:
`visibilità ritorno nomeMetodo`$(U_1, \ldots, U_n)$ `throws` $E_1, \ldots, E_n$

Cosa può cambiare in un overriding e perché?

- La **visibilità** può essere uguale o più ampia, non può essere minore, questo riduce la pre-condizione e rispetta la regola contro-variante
- Il **tipo di ritorno** può variare, il tipo di ritorno può essere un sottotipo dell'originale (Se primitivo non può cambiare), questo rende più specifica la post-condizione e rispetta il principio della sostituibilità, questo si può intendere come un vantaggio in più in quanto non solo ritorna quel tipo ma addirittura una versione più specifica
- Il **nome del metodo**, il numero di parametri ed il loro tipo deve rimanere lo stesso, tecnicamente se un overriding prendesse parametri più generali non violerebbe la regola della sostituibilità, ma per una scelta progettuale non è possibile
- Le **eccezioni** dichiarate con throws possono cambiare in numero e diventare più specifiche ma devono essere sottotipi delle eccezioni del metodo principale, come parte della post-condizione

Si prenda ora il seguente esempio, consideriamo un contratto per il metodo con la seguente intestazione:

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

Pre-condizione: La stringa str non è null e non è vuota, il carattere ch è presente in str.
Post-condizione: Restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str.

Quali dei seguenti contratti sono overriding validi?

1. Pre-condizione: La stringa str non è null
   Post-condizione: Se la stringa è vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str
   Valido: indebolisce la pre-condizione in quanto adesso accetta anche un carattere non presente, così come rafforza la post-condizione con ulteriori casi.

2. Pre-condizione: nessuna
   Post-condizione: se la stringa è null oppure vuota, restituisce 0, altrimenti restituisce il numero di occorrenze di ch in str, diviso la lunghezza di str
   Valido: indebolisce ulteriormente la pre-condizione

3. Pre-condizione: la stringa str non è null e non è vuota
   Post-condizione: restituisce il numero di occorrenze di ch in str
   Non valido: con gli stessi casi leciti cambia radicalmente il valore di ritorno.

11.1.4 Contratti in JavaDoc

JavaDoc è un tool che estrae documentazione dai sorgenti Java e la rende disponibile in vari formati, tra cui l'HTML, JavaDoc estrae informazioni dalle dichiarazioni e da alcuni commenti speciali racchiusi tra /** e */, all'interno di questi commenti è possibile usare tag per strutturare la documentazione, tra i primi qualsiasi tag HTML ma si possono usare alcuni tag speciali come `@param`, `@return` e `@throws`.

Ad esempio, consideriamo un metodo che calcola la **radice quadrata** di un numero dato

- La **pre-condizione** consiste nel fatto che l'argomento deve essere un numero non negativo

- La **post-condizione** assicura che il valore restituito è la radice quadrata dell'argomento
- La **reazione all'errore** 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 quadrata
 * @return la radice quadrata di x
 * @throws IllegalArgumentException se x e' negativo
 */
public double sqrt(double x) {
 if (x < 0) {
    throw new IllegalArgumentException();
 }
 // Il resto della funzione
}
```

11.1.5 Casi particolari di contratti

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

Questa distinzione è importante perché altrimenti equals non rientra in nessun caso in quanto equals nella classe `Object` confronta l'identità e se lo prendiamo come contratto allora dovrebbe confrontare sempre gli indirizzi e non si potrebbe ridefinire equals.

Parte generale che deve essere rispettata da tutti gli overriding:

- *Pre-condizione:* nessuna, tutte le invocazioni sono lecite
- *Post-condizione:*
  - 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 che è riferita esclusivamente per Object:

- *Pre-condizione:* nessuna
- *Post-condizione:* l'invocazione x.equals(y) è equivalente a `x == y` (quando x non sia null)

12 Lezione 12

La lezione 12 mostra i Set e le loro implementazioni, la coerenza tra una relazione d'ordine ed equals ed infine Hashset e la coerenza tra Hashcode ed equals

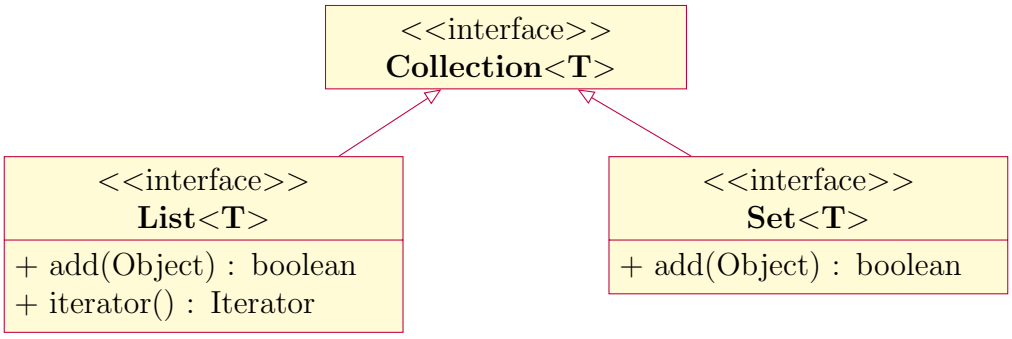
"Invece, oggi il menù prevede: Set e le sue implementazioni"
- M. Faella

12.1 Set ed implementazioni

Abbiamo già visto la struttura delle collezioni già nelle lezioni precedenti, dove Set veniva paragonato ad un insieme matematico, il Set ha due proprietà caratteristiche:

1. L'ordine degli elementi non è rilevante
2. Non accetta duplicati

Analizziamo un po' la differenza tra `List` e `Set` nella JCF:



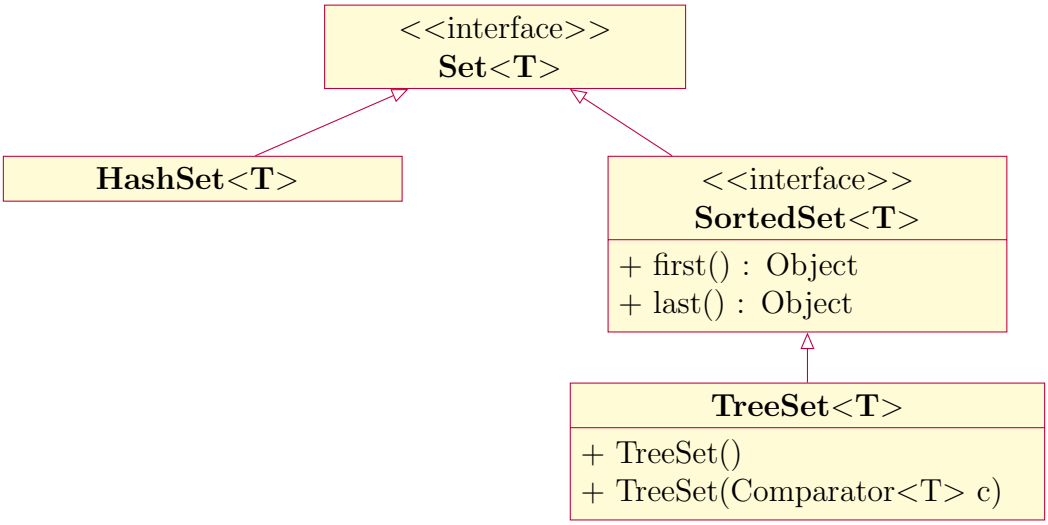
L'interfaccia `Collection` come abbiamo visto contiene i metodi `add`, `remove` `contains`, `iterator` ed altri metodi meno importanti al momento, analizziamo gli overriding dal punto di vista di **Designed by Contract** per l'interfaccia List:

1. Il metodo `add` è compatibile per il contratto che vuole che "Restituisce true se viene inserito l'elemento", per le liste questo è sempre vero
2. Il contratto del metodo `iterator` prevede che l'overriding garantisca un determinato ordine di scorrimento, ovvero secondo l'ordine di inserimento, siccome per le liste l'inserimento è sempre in coda allora è rispettato

Mentre per i set abbiamo che soltanto `add` viene fatto l'override e restituirà true solo se l'elemento non esiste già, non inserendolo altrimenti

12.1.1 Set ed il suo sottoalbero

L'interfaccia set si divide in ulteriori classi ed interfacce che sono le seguenti:



Iniziamo con l'analizzare `SortedSet`, è importante notare che il suo ordinamento è diverso dalle liste, nelle liste l'ordine è relativo all'inserimento, mentre il SortedSet è un insieme i cui elementi sono dotati di una relazione d'ordine specificata tramite `Comparator` o `Comparable`, grazie a questa relazione d'ordine sono implementati i metodi `first()` e `last()` che restituiscono rispettivamente il più piccolo ed il più grande.

Poi abbiamo `TreeSet` che è una classe che estende `SortedSet`, come dice il nome è un albero di ricerca bilanciato, il `TreeSet` supporta due costruttori:

1. Uno senza argomento: Richiede che gli oggetti siano dotati di ordinamento naturale implementando `Comparable`
2. Uno che accetta un comparatore: Gli oggetti possono anche non essere dotati di ordinamento naturale

La ricerca procede per confronti e scende finché non trova l'oggetto oppure arriva alle foglie e quindi restituisce che l'oggetto non c'è, si noti che l'oggetto è identificato tramite comparatore e non tramite equals, a discapito di quanto prevede il contratto di contains della `Collection` che prevede che sia basato su equals, questo però non è un problema in quanto se il comparatore è **coerente** con equals il contratto non è violato (approfondito in 12.1.2).

Infine c'è `HashSet` che è basato sui metodi `HashCode` ed `Equals` in un tabella Hash, questi due richiedono una determinata coerenza tra di loro (approfondito in 12.1.4), la firma di HashCode è la seguente:

```
public int hashCode();
```

I tre metodi principali in TreeSet hanno complessità logaritmica mentre in HashSet è tempo ammortizzato costante, ovviamente questo solo se hashCode riesce a distribuire correttamente i valori nel range degli interi.

12.1.2 Coerenza tra un comparatore ed equals

Un comparatore (sia naturale che non) è coerente con equals se e solo se:
$x.compareTo(y) == 0 \iff x.equals(y)$

Questa proprietà è necessaria affinché `TreeSet` si comporti come previsto dal contratto dell'interfaccia `Collection`, nonostante tutto è possibile violare questo requisito tenendo conto del comportamento che può avere `TreeSet` in questi casi.

12.1.3 Esempio sui TreeSet

Il seguente esempio è basato sull'intercorso del 2017 con "Stanza e Prenotazione", è stato leggermente modificato dal docente rispetto all'originale.

Sia dato il seguente caso d'uso:

```
1 Room r = new Room();
2 Room.Reservation p1 = r.reserve("Pasquale", 105, 120);
3 for(Room.Reservation p : r.reservations()) {
4   System.out.println(p);
5 }
```

Le richieste sono le seguenti:

- Nella riga 2 avviene la prenotazione tramite il metodo `reserve` dal giorno 105 al giorno 120, quindi 15 giorni, questo metodo lancia un eccezione se si prenota da una stanza già occupata ad esempio se si prenotasse da 100 a 106.
- La traccia richiede che vengano stampati in ordine di occupazione e non in ordine di prenotazione, se una persona si prenota per ultima ma per i primi giorni dell'anno dovrà apparire prima.

Le problematiche principali sono accorgersi delle sovrapposizioni e stampare in ordine cronologico, per questo possiamo usare un `TreeSet` che per motivi di *Ingegneria del Software* dichiareremo come di tipo statico `SortedSet`, questo avviene perché la best practice prevede di dichiarare sempre con il tipo più astratto possibile ma che presenta le caratteristiche di cui abbiamo bisogno, in questo caso non usiamo nulla di specifico a `TreeSet` ma solo cose che stanno anche in `SortedSet` ma non lo dichiaro come `Collection` perché ho bisogno del contratto di `SortedSet`, il vantaggio è una forma maggiore di astrazione che permette di cambiare tipo se si dovesse avere un implementazione migliore di `SortedSet`.

Dopo questa osservazione, procediamo con l'implementazione:

```
1  public class Room {
2    private SortedSet<Reservation> reg = new TreeSet<>();
3
4    // Fornisco l'ordinamento naturale che utilizzera' TreeSet
5    public static class Reservation implements Comparable<Reservation> {
6       private int start, end;
7       private String name;
8
9       /* Il costruttore sara' privato perche' sara' il metodo reserve a richiamarlo e
            non vogliamo che venga chiamato dall'esterno */
10      private Reservation(String nome, int start, int end) { /* Solito costruttore*/ }
11
12      public int compareTo(Reservation other) {
```

```
13          // Quando uno dei due inizi sta nell'intervallo dell'altro sono "uguali"
14          if((start >= other.start && start <= other.end) ||
15             (other.start >= start && other.start <= end)) {
16                return 0;
17             }
18
19             /* Se sono qui non si sovrappongono, inoltre siccome i numeri sono in un
                   range ristretto (0 - 365) non corro rischi e posso tornare la
                   differenza */
20             return start - other.start;
21       }
22    }
23
24    public Reservation reserve(String nome, int start, int end) {
25       Reservation p = new Reservation(name, start, end);
26
27       if(!reg.add(p)) {
28          throw new RuntimeException( ... );
29       }
30       return p;
31    }
32
33    /* Ritorno il TreeSet mascherato da Iterable in modo che gli elementi non possano
          essere modificati facilmente, purtroppo Iterable contiene il metodo remove ma
          e' il compromesso migliore*/
34    public Iterable<Reservation> reservations() {
35       return reg;
36    }
37 }
```

In questa implementazione il criterio di ordinamento non è coerente con equals in quanto non ne è stato fatto l'override e viene usato quello di Object, questo non rispetta il contratto di `Collection` ma solo quello di `TreeSet`

12.1.4 Coerenza tra hashCode ed equals

L'HashSet come il TreeSet ha un problema di coerenza, solo che a differenza di quest'ultimo se non viene rispettata la seguente coerenza l'HashSet non funziona:
$x.equals(y) => (x.hashCode() == y.hashCode())$

La sola implicazione è necessaria in quanto i numeri di hashCode sono limitati agli interi rappresentabili rispetto a confronti tra valori i cui confronti sono esponenzialmente maggiori.

Dato il seguente codice, come è necessario ridefinire hashCode affinché sia lecito?

```
1  class Employee {
2    private String name;
3    private int salary;
4    // equals basato su name e salary
5  }
```

Abbiamo varie opzioni:

1. Non ridefinirlo non è possibile, perché 2 employee uguali, con stesso nome e salario, avrebbero hashCode diverso.
2. Ipotizziamo di voler definire hashCode come segue:

```
public int hashCode() {
  return 0;
}
```

   È lecita ma tutti gli elementi finirebbero nello stesso bucket, rendendo l'hash praticamente una lista.

3. Miglioriamolo un po' come segue:

```
public int hashCode() {
  return salary;
}
```

   È coerente, è lecito ed un po' migliore di prima, ma dipendenti con lo stesso salario finiscono nello stesso bucket.

4. La versione migliore però è la seguente:

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

   Utilizziamo il metodo hashCode() delle stringhe, molto elaborato e preciso, da questo facciamo lo XOR bit a bit con il salario, facciamo questo rispetto alla somma per evitare overflow.

Si noti che però se il dipendente avesse avuto un ulteriore campo, ipotizziamo `bonus`, non utilizzato dall'equals, fare un hashCode che ne tenesse conto sarebbe stato illecito in quanto non avrebbe rispettato equals.

12.1.5 Oggetti mutabili e Set

È importante fare attenzione ad inserire in un `TreeSet` o un `HashSet` oggetti mutabili, ovvero oggetti che posseggono metodi di `set`, in quanto la loro modifica dopo un inserimento comporterebbe che la loro posizione non è più corretta nel set.

Si ipotizzi di avere un `TreeSet` di `Employee` ordinati naturalmente per nome, se viene effettuato un set sul nome di un Employee e di conseguenza venga cambiato, questo causa brutte situazioni in cui un Employee non si trova più nella sua posizione corretta, si veda il seguente codice di esempio:

```
TreeSet<Employee> t = new TreeSet<>();
Employee a = new Employee("Luca", 1500);
t.add(a);
// Si ipotizzino altri inserimenti qui
```

```
5 a.setName("Louise");
6 /* Se adesso facessi t.contains(a) mi restituirebbe FALSE, perche' lo cerca in un
     altro nodo dell'albero, paradossalmente sarebbe anche possibile inserire un altro
     elemento a nel TreeSet avendo concretamente duplicati */
```

Di conseguenza se si utilizza un `TreeSet` con classi mutabili è importante che esse non siano modificate mentre sono all'interno del TreeSet, bensì estrarle, modificarle e poi reinserirle.

13 Lezione 13

La lezione 12 mostra il parametro Jolly e le mappe.

"Quest'anno pochi utilizzeranno la prova intercorso in quanto non sembra sia andata bene, forse troppa pastiera o troppo Comicon"
- M. Faella

13.1 Parametro Jolly (Wildcard)

Quando dichiariamo una funzione parametrica non abbiamo sempre bisogno di dichiarare il tipo che prenderà, infatti in alcuni casi in cui questo parametro non viene utilizzato ed una funzione si aspetta un parametro **attuale** di tipo è possibile utilizzare il *parametro Jolly* e si indica con `?`, per chiarire meglio si noti il seguente esempio:

```
1 void fun(List<?> myList)
```

Questa cosa si può leggere come "La funzione fun prende come una lista di tipo sconosciuto" o "una lista di jolly", si noti che fun non è parametrica in quanto non dichiara nessun tipo.

Come abbiamo precisato il Jolly è possibile utilizzarlo unicamente in ambienti che si aspettano un parametro **attuale** di tipo, infatti nei seguenti esempi non è legale in quanto non si sta passando un parametro ma lo si sta dichiarando, vengono segnalati errori di compilazione:

```
1 // Non puo' essere dichiarato prima di una funzione
2 <?> void fun(List<?> myList)
3
4 // Non puo' essere dichiarato in una classe
5 class A<?> {}
```

Passato ad una classe o interfaccia identifica il supertipo comune a tutte le liste, `List<?>` è supertipo di tutte le liste, si può *quasi* intendere come `List<Object>` ma non esattamente in quanto `List<Object>` non è supertipo di tutti.

Proprio per questo è importante notare che facendo una `get` in una lista di Jolly potremo assegnare solo ad `Object` (in quanto qualcosa estratto da una lista sarà sempre un oggetto), ma non è in nessun modo possibile fare la `add` in una lista di jolly eccetto `null` (per le regole di assegnabilità del capitolo 2.4.2).

Si noti il seguente esempio per chiarirne l'utilizzo:

```
1  void f(List<?> myList) {
2    Object x = myList.get(0); // Lecito
3
4    myList.add(x); // Illegale, x deve essere dello stesso tipo della lista, ma la
         lista e' di tipo sconosciuto
5    myList.add(null); // Consentito
6  }
7
8  <T> void f2(List<T> myList) {
9    T t = myList.get(0);
10   myList.add(t);
```

```
}
```

Qui verrebbe da chiedersi quando scegliere un implementazione o l'altra, è importante notare due grosse distinzioni:

1. Nella prima non c'è nessun modo per alterare la lista ed abbiamo una mediocre lettura di essa (otteniamo solo Object).
2. La seconda ci permette di utilizzare la get sapendone esattamente il tipo così come la possibilità di aggiungere elementi.

Continuiamo ad utilizzare il principio del minimo privilegio e se nel corpo non utilizzo il tipo di dato allora favorisco il primo costrutto rispetto al secondo.

13.1.1 Extends e Super di Jolly

Come visto precedentemente un parametro di tipo può essere limitato superiormente come segue:

```
<T extends Employee> void f(List<T> myList) {}
```

Siccome un parametro Jolly non viene dichiarato rispetto ad uno di tipo, questa cosa avviene esplicitamente nel momento del suo utilizzo:

```
void f(List<? extends Employee> myList) {}
```

Si noti che le regole del Jolly variano quando esso ottiene un limite superiore, infatti sapendo che `<? extends Employee>` sappiamo che il Jolly sarà necessariamente una sottoclasse di Employee e quindi assegnabile direttamente a lui, però rimane sempre impossibile fare assegnamenti in quanto non sappiamo se la lista è di Manager o Employee, aggiungere `null` rimane ancora valido.

Vediamo un esempio con parametro di tipo e Jolly:

```
<T extends Employee> void f(List<T> myList) {
  T t = myList.get(0);
  System.out.println(t.getName());
}

void f2(List<? extends Employee> myList) {
  Employee x = myList.get(0); // ? e' una sottoclasse di Employee
  myList.add(x); // Errore di compilazione, ? puo' essere Employee ma anche Manager
  myList.add(new Manager(...)); // Stesso motivo di sopra
  myList.add(null); // Valido
}
```

Il limite superiore sul Jolly permette di leggere con maggiore precisione ma allo stesso tempo impedisce di poter aggiunger elementi che non siano null, quindi abbiamo una lettura migliore con un ulteriore garanzia che la lista non può essere alterata (oltre l'aggiunta di null).

A differenza del parametro di tipo il Jolly permette di avere anche un limite inferiore "riciclando" la keyword `super` ed utilizzandola come segue:

```
void f(List<? super Employee> myList)
```

Con `super` abbiamo garanzie opposte rispetto ad `extends`, ovvero la `get` si può assegnare unicamente ad Object ma permette di aggiungere elementi alla lista, questo può sembrare strano in quanto la keyword permette di assegnare tutti gli elementi di quel tipo e le relative **sotto**classi ma non **super**classi, ovvero ha un significato opposto al suo nome, si noti il seguente esempio:

```
void f(List<? super Employee> myList) {
  Object x = myList.get(0); // Non posso assegnare
  myList.add(new Employee(...)); // Per il polimorfismo posso mettere Manager anche
      in liste di Employee
}
```

Si ipotizzino le seguenti firme:

```
void f(Set<?> a, Set<?> b);
<T> void f(Set<T> a, Set<T> b);
```

Si noti che nel secondo caso accetteremo solo due Set dello stesso tipo, mentre la prima consente due Set di qualsiasi tipo, anche diversi tra loro, infatti con il Jolly non è possibile forzare 2 parametri dello stesso tipo in ingresso.

Se un metodo utilizza un solo parametro di tipo che però non compare più nel corpo e non viene ritornato, allora è possibile convertire il metodo in non-parametrico sfruttando uno Jolly, come ad esempio:

```
<S> void g(Collection<S> c); // Si puo' convertire se non uso S nel corpo
void g(Collection<?> c);
```

Invece se il tipo parametrico compare nel ritorno non è possibile utilizzare Jolly, il valore in ingresso è legato con quello di ritorno:

```
<S> S g(Collection<S> c);
```

13.1.2 Esempi con Jolly

Si ipotizzi di voler scrivere un metodo che accetta oggetti dotati di ordinamento naturale e ne restituisca il minimo, si noti il seguente scenario:
Employee implementa Comparable<Employee> ed ha una sottoclasse Manager, in maniera indiretta allora anche Manager implementa Comparable.

Analizziamo varie possibili firme per questo metodo:

1. `Comparable getMin(Collection<Comparable> c)`
   Non va bene, utilizza la versione grezza di Comparable
2. `<T> Comparable<T> getMin(Collection<Comparable<T>> c)`
   Non funziona perché non accetta `Collection<Employee>` in quanto non è assegnabile ad Employee (Vedesi l'assegnabilità dei tipi parametrici a capitolo 8.4)
3. `<T extends Comparable<?>> T getMin(Collection<T> c);`
   Se T = Employee, soddisfa il limite superiore e posso assegnarlo anche a Collection<Employee>
   Se T = Manager, soddisfa il limite superiore e siccome Employee implementa Comparable posso assegnarlo, quindi è la firma migliore

Procediamo ad implementarlo:

```
1  <T extends Comparable<?>> T getMin(Collection<T> c) {
2    T min = null;
3
4    for(T t : c) {
5       if(min == null || min.compareTo(t) > 0) {
6          min = t;
7       }
8    }
9    return min;
10 }
```

Possiamo vedere subito un grave errore alla riga 5, infatti la chiamata al metodo `compareTo` è illecita, perché potrei passargli solo null, è un metodo analogo ad add in una lista e quindi a `Comparable<?>` non posso passare altro che null, questo si risolve cambiando la firma come segue:

```
1  <T extends Comparable<? super T>> T getMin(Collection<T> c);
```

13.2 Array associativi (Mappe)

È una struttura dati ed in Java parte da un interfaccia `Map<K, V>`, si può notare che ha due parametri di tipo di cui il primo è il tipo della chiave (deve essere univoca) ed il secondo è il tipo del valore, le mappe non sono iterabili in quanto non estendono Iterable e non sono imparentate con Collection.

I metodi dell'interfaccia sono:

- `+put(K, V) : V`
  Prende in ingresso una chiave ed un valore, restituisce null o un valore nel caso la chiave fosse già precedente e lo abbia sovrascritto
- `+get(Object) : V`
  Prende una chiave dichiarata come Object, questo è utile per i metodi che utilizzano parametri Jolly e non è rischioso
- `+size() : int`
  Banalmente il numero di chiavi
- `+remove(Object) : V`
  Rimuove l'elemento con la chiave che corrisponde ad Object e lo restituisce
- `+KeySet() : Set<K>`
  Restituisce una vista sull'insieme delle chiavi, ovvero è leggibile ma non modificabile, è la mappa stessa che presenta delle funzionalità di lettura e lancia eccezioni sulle funzioni di modifica
- `+values() : Collection<V>`
  Restituisce una lista, come il caso precedente, di sola lettura dei valori della mappa, è una collezione perché rispetto alle chiavi i valori possono avere duplicati

C'è una forte analogia con i `Set` in quanto `Map` ha due sottoclassi che sono `HashMap` e `TreeMap`, questo perché gli `HashSet` e `TreeSet` sono implementati proprio utilizzando queste ultime due.

`HashMap` mette nel bucket in base all'HashCode della chiave, come per i `Set` c'è la stessa gestione per le chiavi mutabili.

`SortedMap` in analogia con `SortedSet` che aggiunge i metodi `first` e `last`, aggiunge i metodi che restituiscono la chiave minore e massima tramite Comparable o Comparator, i metodi sono i seguenti:

- `+firstKey() : K`
- `+lastKey() : K`

13.2.1 Esercizio su Mappe e Jolly

Scrivere un metodo che restituisce se una mappa data in ingresso è iniettiva.

```
<K, V> boolean isInjective(Map<K, V> map) {
  Set<V> set = new HashSet<>();

  for(V val : map.values()) {
     if(!set.add(val)) {
        return false;
     }
  }
  return true;
}
```

Da qui possiamo notare che K compare una volta sola nel metodo, non c'è bisogno di dichiararla, quindi possiamo cambiare la firma come segue:

```
<V> boolean isInjective(Map<?, V> map);
```

Però non è finita qui, in realtà possiamo cambiare l'intero metodo per renderlo non parametrico in quanto non ci serve sapere il tipo di V:

```
boolean isInjective(Map<?, ?> map) {
  Set<Object> set = new HashSet<>(); // Non sappiamo V, ma sara' almeno Object

  for(Object val : map.values()) {
     // Valido in quanto set e' un Set<Object>
     if(!set.add(val)) {
        return false;
     }
  }
  return true;
}
```

13.2.2 Esercizio 2 su Mappe

Data una collezione di numeri si vuole ottenere la loro somma.

```
double getSum(Collection<? extends Number> c) {
```

```
2     double sum = 0;
3     for(Number n : c) {
4        sum += n.doubleValue();
5     }
6     return sum;
7  }
```

Scegliamo un double perché non c'è modo di tornare lo stesso tipo di quello in ingresso in quanto non esiste aritmetica tra i wrappers.

14 Lezione 14

***Nota**: Questa lezione è abbozzata da appunti presi da alcuni colleghi in quanto quel giorno ero assente, quindi è meno dettagliata rispetto le precedenti ma ho cercato di fare del mio meglio.*

14.1 Implementazione dei Generics

I Generics sono presenti in molti linguaggi con numerosi pro e contro, vediamo le implementazioni in `C++`, `C#` e `Java`, con reificazione intendiamo rendere concreto il tipo di dato:

C++ (Template)	C# (Generics)	Java (Generics)
Implementati con reificazione a tempo di compilazione	Implementati con reificazione a runtime	Implementati a cancellazione (erasure)
Concreti in compilazione	Ibrido tra C++ e Java	Implementazione più astratta e leggera tra le tre
	Approccio possibile perché C# ha una VM quindi completo controllo sullo stack di esecuzione, dal template produce un oggetto speciale che rappresenta una classe parametrica (C++ non può farlo), questa classe è pre compilata e on-demand ne vengono create istanze	
PRO: Estremamente potente, con il tipo posso fare qualsiasi cosa ed una volta compilato diventa un tipo concreto. Posso fare `new T()` (che in Java dopo compilazione T diventa Object), i metodi specifici del tipo T restano comunque non utilizzabili	**PRO**: Mantiene il vantaggio del c++ facendo diventare T concreto	**PRO**: Non ha overhead su tempo e spazio
CONTRO: Overhead sulla dimensione del codice (nel codice oggetto del programma mi trovo n copie molto simili di LinkedList)	**CONTRO**: Overhead di tempo durante le esecuzioni	**CONTRO**: molte limitazioni su ciò che è possibile fare con T
CONTRO: Problemi con la compilazione separata di vari file sorgente appartenenti allo stesso progetto, viola il principio di compilazione separata, per risolvere la classe template viene messa in un .h separato che verrà implementato nei .cpp. Questo ha portato alla diffusione di librerie "header only": Non sono precompilate ma sono tutta l'implementazione è da includere		

14.1.1 Erasure

Vediamo ora nel dettaglio l'implementazione dei Generics in Java:

1. Generics usati per il type checking
2. I parametri formali di tipo (T, S, V, *etc.*) vengono rimossi e sostituiti con il loro primo limite superiore (oppure Object)
   Ad esempio <T extends Comparable<T> & ... >, tutte le occorrenze di T vengono sostituite con Comparable
3. Vengono inseriti i cast opportuni
   Ad esempio ogni volta che viene utilizzato un metodo che restituisce T
4. I parametri attuali jolly vengono rimossi semplicemente
5. Metodi ponte (Non trattati nel corso)

14.1.2 Conseguenze dell'erasure

Seguono le conseguenze per le regole precedentemente stabilite:

- `new T()` Errore di compilazione, siccome T è un tipo virtuale non arriva a runtime, da non confondere con `new LinkedList<T>` che è OK
- `new LinkedList<?>()` Errore di compilazione, non ha senso
- `new T[10]` errore di compilazione, gli array ricordano a runtime il loro tipo di creazione, ma siccome T a runtime è cancellato porta comportamenti non previsti dal programmatore
- `T[] array` OK, array di Object

14.1.3 Cast e instanceof

Abbiamo un comportamento particolare per quanto riguarda i cast ed istanceof dei generics:

- `(T) exp`, un cast a T, sconsigliato e viene segnalato un warning
- `(Lista <String>) myList`, sconsigliato, è equivalente a scrivere `(List) l` e viene segnalato un warning
- `(List<Object>) new Object()`, a compilazione da un warning ma lancia `ClassCastException` a runtime
- `(List<String>) new LinkedList<Integer>()` Errore di compilazione
- `(List <String>) (List<?>) new LinkedList<Integer>()`, segnala un warning ma non da problemi a runtime, si noti che però se si fa una get e la si assegna ad una stringa viene lanciata un eccezione
- `exp istanceof T` errore di compilazione poiché a runtime è stato cancellato
- `exp istanceof List<String>`

14.2 Riflessione

Si basa su una classe speciale managed di nome `Class`, la sua dichiarazione:

```
class Class <T> {
  public String getName();
  public T newInstance(); // Richiama il costruttore vuoto della classe
  public Class<? super T> getSuperClass() // Superclasse diretta
  public boolean isInstance(Object x); // Ritorna true se il tipo effettivo di x e'
      sottotipo, si puo' vedere come una versione dinamica di instanceof
}
```

In `Object` inoltre abbiamo il metodo:

```
public Class<?> getClass()
```

Questo fa eccezione al comportamento normale in quanto poi quando si crea una nuova classe A diventa `<? extends A>`, si veda il seguente esempio:

```
f(Person p {
  Class<? extends Person> c = p.getClass();
}
```

15 Lezione 15

La lezione 15 continua con la riflessione, ne fa un confronto con i Generics ed inizia con la scelta dell'intestazione migliore per un metodo.

15.1 Continuo sulla Riflessione

Di questi meccanismi il `C` non ne ha nessuno, il `C++` ne ha alcuni mentre `Java` ne è estremamente ricco, molti di essi sono contenuti nella classe `Class`.

Come già visto la classe `Class` è parametrica, presa un ipotetica classe `Employee` il suo oggetto di tipo Class avrà la forma `Class<Employee>`.

Per accedere a questa classe ci sono due modi, il primo è invocare il metodo `getClass()` su un istanza di classe, il secondo è utilizzare l'operatore statico `.class` sul nome della classe, ad esempio:

```
1 Employee e = new Employee(...);
2 c = Employee.class;
3 c2 = e.getClass();
```

Nel primo caso il compilatore sa chiaramente che `c` è un Employee, mentre nel secondo caso il compilatore sa che sarà *almeno* un Employee, infatti possiamo stabilire che:

- L'oggetto Class di c sarà `Class<Employee>`
- L'oggetto Class di c2 sarà `Class<? extends Employee>`

Nonostante questa differenza `c == c2` restituirà `true` in quanto per ogni classe esiste un solo ed unico oggetto Class.

Vediamo adesso una serie di combinazioni con `getClass()` e `.class`:
Data la dichiarazione `Employee e = new Manager(...);`

1. e instanceof Employee
2. e instanceof Manager
3. e.class instanceof Employee
4. e.getClass() == Manager
5. e.getClass() == Employee.class
6. e.getClass() == "Manager"
7. e.getClass() == Manager.class
8. e.getClass().equals(Manager.class)

Analizziamoli uno ad uno:

1. True, il tipo non è null ed il tipo effettivo di `e` è un sottotipo di Employee
2. True, stesso motivo precedente
3. Errore di compilazione, `e.class` non è un espressione valida perché l'operatore `.class` accetta **solo** il nome di una classe

4. Errore di compilazione, Manager non è un espressione valida, non si può utilizzare il nome di una classe senza niente con l'operatore `==`
5. False, espressione valida ma oggetti diversi
6. In maniera inaspettata errore di compilazione, questo a causa di una scelta implementativa in quanto il confronto non supera il type checking, le due espressioni non hanno nessuna parentela, a sinistra c'è un oggetto `Class` ed a destra uno `String`
7. True
8. True, coincide con il punto 7 in quanto l'`equals` o è quello di Object (allora puntano allo stesso oggetto Class) oppure per riflessività

Applichiamo ora questi concetti in un esercizio, ipotizziamo di voler creare un metodo `Fill` che abbia il seguente caso d'uso:

```
Employee[] a = new Employee[20]; // Inizializzato con 20 null
Test.<Employee>fill(a, Employee.class); // Riempie l'array di Employee
```

La sua implementazione potrebbe essere la seguente:

```
public static <T> void fill(T[] array, Class<T> c) throws Exception {
  for(int i = 0; i < array.length; i++) {
     array[i] = c.newInstance();
  }
}
```

Si noti che i metodi della classe `Class` lanciano una serie di eccezioni verificate che devono essere segnalate, per colmare questo aggiungiamo `throws Exception`, si noti inoltre che ingenuamente si potrebbe voler usare un enhanced for, ma si ricordi che questo costrutto non può modificare l'array ed è quindi necessario un for tradizionale (si veda 9.1).

Supponiamo di voler cambiare il caso d'uso:

```
Employee[] a = new Employee[20]; // Inizializzato con 20 null
Test.<Employee>fill(a, Manager.class); // Riempie l'array di Employee
```

In questo caso l'implementazione non va più bene a causa dell'intestazione del metodo, infatti questa intestazione non è completa rispetto al contratto di `fill`, ovvero non accetta tutti i parametri possibili (in questo caso Manager), procediamo a renderla valida:

```
public static <T> void fill(T[] array, Class<? extends T> c) throws Exception
```

Questa modifica fa sì che all'esterno funzioni, dall'interno invece abbiamo che `c.newInstance()` restituisce `? extends T`, il valore di ritorno si può assegnare a T e suoi supertipi, quindi funziona, con questa modifica l'intestazione è completa rispetto al seguente contratto di `fill`: *"Accetta un array ed una classe tale che la classe è sottotipo dell'array"*

Con la riflessione è possibile fare molte cose, come abbiamo già visto il metodo `newInstance()` permette di creare istanze di una classe richiamando il costruttore vuoto, ma ci sono altri metodi che permettono di scoprire attributi, metodi e moltre altre caratteristiche di una classe per poterli utilizzare a runtime, vediamone alcuni:

```java
// In Class
public Field[] getFields(); // Restituisce i campi pubblici di questa classe, anche
    quelli ereditati dalla superclasse, ma sempre pubblici
public Field[] getDeclaredFields(); // Restituisce anche i campi non pubblici
// Metodi analoghi ma per i metodi della classe
public Method[] getMethods();
public Method[] getDeclaredMethods();
```

Come si può vedere i primi due metodi ritornano oggetti di tipo `Field`, vediamo che metodi possiede questa classe:

```java
// In Field
public String getName(); // Nome del campo
public Object get(Object x) throws IllegalAccessException; // Restituisce il valore
    del campo dell'oggetto x, se non e' statico allora sara' diverso per ogni istanza
public Class<?> getType();
public void set(Object x, Object val);
```

Tutti questi metodi sono controllati dal *Security Manager* della JVM, di default essa impedisce di leggere e scrivere su campi privati ma è possibile modificare il flag per ottenerne il permesso.

Oltre la classe `Field` abbiamo quella `Method`, che però è analoga a questa eccetto il metodo `invoke` che permette di invocare il metodo passando in ingresso i vari parametri che si aspetterebbe.

Sebbene possa sembrare una funzionalità inutile, la riflessione è essenziale per alcuni framework esterni come Ibernate, Spring, etc.

15.1.1 Confronto tra Riflessione e Generics

Torniamo al primo esempio visto con i Generics:

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

Supponiamo di volerlo realizzare senza generics ma con il medesimo funzionamento, ovvero con gli stessi benefici di controllo di tipo, allora possiamo realizzarlo come segue:

```java
class Pair {
  private Class type;
  private T first, second;
  public Pair(Class c, Object a, Object b) {
     // Prima lo faceva il compilatore, ora e' essenziale a runtime
     if(!c.isInstance(a)) {
        throw new IllegalArgumentException();
     }
  }
}
```

Abbiamo che questa implementazione è più potente in quanto viene ricordato il tipo, però perde totalmente il vantaggio dei Generics, ovvero quello di far apparire mismatch di tipo in compilazione.

15.2 Scelta dell'intestazione migliore (Parametri Formali)

La scelta dell'intestazione migliore si divide in due parti, quella dell'analisi dei parametri formali e quella del tipo di ritorno, in questo capitolo parleremo del primo caso.

La scelta dei parametri formali è relativo al contratto, ovvero devono accettare i requisiti della pre-condizione e rifiutare quelli che non la rispettano, proprio per questo differenziamo ben 5 criteri di scelta, in ordine di importanza:

- *Funzionalità*: Consente di scrivere il corpo del metodo senza usare cast superflui, è la più importante in quanto in sua mancanza non ha senso controllare i punti successivi
- *Completezza*: Accetta tutti gli argomenti "buoni" del contratto
- *Correttezza*: Rifiuta tutti gli argomenti "cattivi" del contratto
- *Ulteriori garanzie*: Offre altri vantaggi, come ad esempio la non modifica di liste in ingresso
- *Semplicità*: Criterio di confronto rispetto ad un'altra intestazione, non ha senso applicarlo su una singola firma, indica se una firma ha meno parametri formali e quindi risulta più "semplice"

Vediamo subito con alcuni esempi, si prendano i seguenti contratti:

1. *Pre-condizione:* Accetta una qualsiasi lista non vuota
   Come parametri potremmo prendere `List<?>` ma purtroppo non esiste un intestazione che mi assicuri che la lista non sia vuota, questa intestazione è quindi *completa* ma non *corretta*, tecnicamente si potrebbe rendere anche corretta creato una classe `NonEmptyList<T>` che estende `List<T>` ma non possiede il costruttore vuoto.
2. *Pre-condizione:* Accetta un intero non negativo
   Come parametri possiamo prendere un int per renderla *completa* ma non sarà *corretta* per l'assenza degli `unsigned int` in Java.

In generale è meglio scegliere un intestazione completa piuttosto che corretta, se proprio non posso averle entrambe, e successivamente fare controlli in runtime per far rispettare il contratto.

15.2.1 Esempio sulla scelta dei parametri formali

Si ipotizzi il metodo `void addAll(src, dest)` definito col seguente contratto:

- *Pre-condizione*: accetta 2 collezioni tali che gli oggetti della prima si possono inserire nella seconda
- *Post-condizione*: dopo l'esecuzione la seconda collezione contiene tutti gli elementi precedenti all'esecuzione più tutti quelli della prima e la prima collezione non viene modificata

Questo contratto si può leggere più formalmente come segue:
$dest' = dest \cup src$
$src' = src$

Valutiamo adesso varie possibili intestazioni per il seguente contratto:

1. `void addAll(Collection<?> src, Collection<?> dest);`
   A primo impatto la firma è completa ma non corretta, ma si è saltata l'analisi più importante, ovvero che questa firma **non è funzionale**, avendo gli jolly la seconda lista non è modificabile.

2. `<T> void addAll(Collection<T> src, Collection<T> dest);`
   Funzionale, corretta, non completa, per dimostrare la mancanza di completezza si noti il seguente contro esempio:
   src = `Collection<Manager>`
   dest = `Collection<Employee>`

3. `<S, T extends S> void addAll(Collection<T> src, Collection<S> dest);`
   La prima firma funzionale, completa e corretta.

4. `<T> void addall(Collection<T> src, Collection<? super T> dest);`
   Funzionale, completa, corretta e più semplice della 3, è la stessa della 3 ma è migliore perché usa un parametro di tipo in meno.

5. `<T> void addAll(Collection<? extends T> src, Collection<T> dest);`
   Funzionale, completa, corretta e più semplice della 3 ma stessa semplicità della 4.

6. `<T> void addAll(Collection<? extends T> src, Collection<? super T> dest);`
   Funzionale, completa, corretta e migliore della 4 e 5 in quanto garantisce al chiamante che può solo leggere da src e scrivere in dest, quindi *gode di ulteriori garanzie*.

Non a caso nella classe `Collections` troviamo il seguente metodo:

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

Che si sposa perfettamente con la firma trovata per `addAll`.

16 Lezione 16

La lezione 16 approfondisce un po' la continua con la scelta dell'intestazione, ne fa un esempio ed introduce le enumerazioni.

16.1 Scelta dell'intestazione migliore (Tipo di ritorno)

La scelta del tipo di ritorno è un po' più delicata rispetto alla lista di criteri che ci si trova per la scelta dei parametri formali, si prenda il seguente esempio:
Ipotizziamo di avere un metodo che ritorna `LinkedList<Employee>`, possiamo dichiarare il tipo di ritorno in vari modi, ad esempio:

- `Object` *(Più astratto)*
- `Collection<Employee>`
- `List<?>`
- `List<Employee>`
- `LinkedList<Employee>` *(Più specifico)*

Ovviamente la scelta del migliore dipende dal contesto, però in generale:

1. *Più specifico:* Vogliamo dare al chiamante più informazioni possibili su quell'oggetto
2. *Più astratto:* Vogliamo limitare il chiamante, oppure sfruttare l'incapsulamento nel caso di evoluzione del software, ovvero scelgo il tipo più astratto in modo che modificando quello specifico non si incappa in problemi futuri.

Una regola generale se ci si trova in stallo è quella di restituire l'interfaccia più specifica, così il chiamante ha il massimo delle informazioni senza divulgarne la classe concreta, in questo caso la scelta migliore sarebbe `List<Employee>`

16.1.1 Esempio sulla scelta dell'intestazione migliore

Molto spesso in questo tipo di esercizi viene dato un contratto informale, come si potrà vedere lo svolgimento richiede in maniera velata di farne una versione più formale in modo che la scelta sia correttamente giustificata.

Sia dato il seguente contratto informale: *Un metodo che prende 2 Set e restituisce la loro intersezione*
Da questo contratto non è chiaro se vogliamo che i Set siano dello stesso tipo o meno, sappiamo solo che vogliamo prendere Set che *potrebbero* avere oggetti in comune, ma purtroppo Java non permette di rappresentare questa condizione, potremmo prendere il secondo Set come sottotipo del primo, ma ci troveremmo in una situazione bizzarra dove:
`Set<Comparable<?>>`
`Set<Iterable<?>>`
Non sarebbero parametri validi, però due classi separate possono implementarli, di conseguenza il nostro **contratto formale** sarà il seguente: *Il metodo accetta 2 set qualsiasi e ne restituisce la loro intersezione*

Da qui possiamo passare alla scelta dell'intestazione migliore tra le seguenti:

1. `<T> Set<T> intersection(Set<T> a, Set<T> b);`
   Funzionale ma non completa per il contratto stabilito.

2. `<T> Set<T> intersection(Set<T> a, Set<? extends T> b);`
   Funzionale ma non completa (Il caso analizzato precedentemente con Comparable ed Iterable)

3. `Set<?> intersection(Set<?> a, Set<?> b);`
   Funzionale, posso ritornare un `HashSet<Object>`, ciclare su uno dei due Set, utilizzare containts e poi add in quanto questi ultimi metodi accettano `Object` come parametri.
   È completa, corretta ed offre garanzie (non posso inserire in a e b), non verifichiamo la semplicità in quanto è la prima completa e corretta.
   Mentre per il tipo di ritorno è discreta dal punto di vista dell'evoluzione ma pessima dal punto di vista del chiamante, si è perso il tipo del Set ed è necessario fare numerosi cast per accedervi, il tipo di ritorno è **troppo generico.**

4. `<S, T> Set<S> intersection(Set<S> a, Set<T> b);`
   Funzionale grazie alla permissività di Contains e Add, completa, corretta, non offre garanzie, meno semplice della 3.
   Il tipo di ritorno è un grosso passo in avanti rispetto la 3, concretamente solo questo la rende migliore della 3.

5. `<S> Set<S> intersection(Set<S> a, Set<?> b);`
   Funzionale, completa, corretta, offre medie garanzie, più semplice della 4 e risolve il problema di T inutilizzato, il tipo di ritorno è gestito dal chiamante, mettendo come primo parametro si ritroverà quello stesso tipo di Set come ritorno.

6. `<T> Set<T> intersection(Set<? extends T> a, Set<? extends T> b);`
   Funzionale, completa, corretta, offre maggiori garanzie su a ma minori su b rispetto la 5, più semplice della 5, anche qui perdiamo informazioni sul tipo di dato in ingresso, se passassi Iterable e comparable avrei un `Set<Object>` come ritorno

Concludiamo quindi che la 5 è la migliore, non a caso la si vede anche in Google Guava (Libreria open-source Google per collezioni di Java), nella loro classe Sets:

```
1 public static <E> Sets.SetView<E> intersection(Set <E> a, Set<?> b);
```

16.2 Enumerations

Si ipotizzi di voler creare uno schedule con il seguente caso d'uso:

```
1 Schedule s = new Schedule();
2 s.addClass("venerdi", 11, 13, "LP2");
```

Possiamo notare che c'è un pericoloso problema di ambiguità per la codifica, ovvero devo mettere "Venerdì", "Venerdi", "venerdi" o "venerdì"? Oppure devo scriverlo in un'altra lingua? Abbiamo un dominio troppo grande per memorizzare tutto sommato solo 7 tipi di stringhe.
Per casi come questi, dove sappiamo che i possibili valori sono pochi e sempre quelli, si utilizzano le enumerazioni, ma prima di introdurle vediamo come implementarle dove non sono supportate.

16.2.1 Typesafe enum pattern

Il TEP è un pattern implementativo per implementare le enumerazioni in linguaggi che non le supportano, ipotizziamo di creare la classe `WeekDay`:

```
class WeekDay {
  private int position;
  // Elimino il costruttore di default creandone uno privato
  private WeekDay(int pos) { ... }
  public static final WeekDay MONDAY = new WeekDay(0);
  ...
}
```

Con questa classe è possibile avere il seguente caso d'uso:

```
s.addClass(WeekDay.FRIDAY, 11, 13, "LP2");
```

L'ambiguità è ora gestita dal compilatore che si occupa di segnalarla ed il dominio è limitato in quanto WeekDay non ha costruttori pubblici.

16.2.2 Implementazione delle enumerazioni

Implementiamo ora `WeekDay` con le enumerazioni:

```
enum WeekDay {MONDAY, TUESDAY, ..., FRIDAY;}
```

Questa si può vedere implicitamente come segue in quanto tutte le enumerazioni implicitamente estendono la classe `Enum`

```
class WeekDay extends Enum<WeekDay>
```

Una caratteristica particolare delle enumerazioni è che ad ogni valore di esse viene associato un valore corrisponde un ordinale, un po' come l'attributo "position" nella classe con il *Typesafe enum pattern*, oltre questo ordinale ogni valore dell'enumerazione ha associato una stringa equivalente, ad esempio `MONDAY` ha come ordinale associato `0` e come stringa `"MONDAY"`.

La classe `Enum` offre vari metodi che consentono di passare da un valore dell'enumerazione ad ordinale o stringa e viceversa, i metodi sono i seguenti nella classe Enum:

```
public int ordinal();
public String name();
// Da stringa a valore
public static <T extends Enum<T>> T valueOf(Class<T> c, String name);
// Da ordinale a valore
public static WeekDay[] values();
```

Il passaggio da nome ad istanza dell'enumerazione richiede proprio che `Class` sia un enumerazione in quanto nessuna classe può estendere `Enum` (oltre le enumerazioni in modo implicito).

Si noti il seguente esempio di utilizzo per ottenere il valore `MONDAY`, si noti che per il primo metodo si ritorna alla problematica dell'ambiguità in quanto "MONDAY" è diverso da "monday" e così via.

```
WeekDay a = Enum.<WeekDay>valueOf(WeekDay.class, "MONDAY");
WeekDay b = WeekDay.values()[0];
```

17 Lezione 17

La lezione 17 continua con le enumerazioni, mostra la differenza tra Costruttori e Factory Methods, il rapporto tra Enumerazioni e Collection, l'esercizio subMap ed introduce i Threads

17.1 Continuo sulle enumerations

Anche il `C` ha un minimo supporto alle enumerazioni, ma questo è relegato solo al dare un valore numerico ai vari campi, come visto precedentemente il supporto di Java è molto più esteso.

Riprendiamo con l'esempio di WeekDay, ipotizziamo di voler aggiungere un metodo che ci dica se un giorno è feriale o meno (Torna false per sabato e domenica, true altrimenti), con il seguente caso d'uso:

```
f(WeekDay d) {
  if(d.isWorkDay()) {
  ...
  }
}
```

Abbiamo due possibili implementazioni, per la prima aggiungiamo un attributo booleano in WeekDay ed un costruttore:

```
enum WeekDay {
  MONDAY(true), TUESDAY(true), ..., SUNDAY(false);

  private boolean workDay;
  public boolean isWorkDay() { return workDay; }
  // Costruttore privato, imposto dalla classe Enum
  private WeekDay(boolean wd) { workDay = wd; }
}
```

Come si può notare è necessario un costruttore esplicito in quanto non possiamo sapere quali giorni siano feriali e non, inoltre rispetto ad una classe classica il costruttore viene scritto vicino il loro valore dell'attributo e viene così "inizializzato", ovviamente è possibile fare overloading di costruttori come si fa usualmente.

Il secondo metodo non prevede costruttore né attributo, semplicemente dichiariamo un metodo e ritorniamo il valore più comune alle istanze (true in questo caso) mentre per quelli meno comuni ne facciamo l'overriding simile alle classi anonime:

```
enum WeekDay {
  MONDAY, TUESDAY, ..., SATURDAY {
     public boolean isWorkDay() { return false; }
  }, SUNDAY {
     public boolean isWorkDay() { return false; }
  };

  public boolean isWorkDay() { return true; }
}
```

Questo è più che altro *syntactic sugar* infatti seguendo il typesafe enum pattern "dietro le quinte" viene scritto e compilato come segue:

```
public static final WeekDay SATURDAY = new WeekDay() {
  public boolean isWorkDay() { return false; }
}
```

Si noti che le enumerazioni sono `final` e non possono essere estese, infatti `SATURDAY` è public ma anche final, non vogliamo mai attributi pubblici liberamente modificabili, ma solo costanti pubbliche.

17.2 Costruttori vs Factory Methods

Anche se non esplicitato abbiamo già visto alcune classi che offrono *factory methods* per creare istanze di una classe, un esempio è la classe Integer:

```
Integer n = new Integer(42);
Integer n = Integer.valueOf(42);
Integer n = 42;
```

Di questi `valueOf` è il factory method in quanto è statico e restituisce un istanza di quell'oggetto, come già parlato durante l'auto-boxing (Capitolo 3.2.1) viene utilizzato un sistema di caching che è più formalment chiamato **memoization**.

La prima differenza sostanziale è che un factory method non è obbligato a ritornare sempre un nuovo oggetto rispetto alla keyword `new`, inoltre l'oggetto creato con la new sarà sempre quell'oggetto specifico, un `new Integer` crea un Integer mentre un factory method può tornarne anche una sua sottoclasse (per il principio di sostituzione funzionerà comunque).

17.3 Enum e Collections

Le enumerazioni e le collezioni hanno un particolare rapporto, implementano i soliti metodi di `Set` e `Map` ma hanno un modo di essere costruite differente, le classi sono:

- `EnumSet<T>`, un implementazione di Set che richiede che `<T>` sia un enumerazione
- `EnumMap<K, V>`, un implementazione di Map che richiede che `K`, le chiavi, siano enumerazioni

17.3.1 EnumSet ed EnumMap

Ipotizziamo di voler creare un `EnumSet<WeekDay>`, internamente quest'ultimo è implementato come un **bitset**, ovvero vengono utilizzati un numero finito di bit dove ognuno di essi rappresenta la presenza o meno dell'*i-esimo* elemento dell'enumerazione, in questo caso sarebbero 8 bit (si va a potenze di 2).

Per creare un EnumSet viene utilizzato il seguente metodo di fabbrica:

```
public static <T extends Enum<T>> EnumSet<T> noneOf(Class<T> c);
```

Come già visto la caratteristica principale di un metodo Factory è di poter restituire un vecchio set oppure cambiarne il tipo, in questo caso non ha senso restituire un vecchio set in quanto non

è immutabile, però sfrutta il secondo vantaggio, dichiararlo come tipo diverso.
Infatti internamente questo metodo restituisce un long (per enumerazioni minori di 64 valori) oppure un array per enumerazioni più lunghe, questa distinzione è fatta per ottenere maggiori performance e supportare qualsiasi tipo di enumerazione, inoltre:

> *"Copre il 99% delle enumerazioni che un programmatore sano di mente utilizzerebbe"*
> *- M. Faella*

Il metodo `noneOf` prende un parametro `Class` per l'erasure dei Generics, tramite questo parametro è possibile fare riflessione e scoprire il numero delle istanze dell'enumerazione.

Il nome `noneOf` si riferisce al fatto che restituisce un insieme vuoto, il suo opposto è `allOf` e restituisce un insieme pieno con tutti i valori dell'enumerazione.

Banalmente si potrebbe implementare lo stesso funzionamento con un `HashSet` ma è considerato *Code smell*, non ha senso implementare da sé qualcosa quando c'è già una versione migliore a disposizione.

Invece le `EnumMap` si creano con un costruttore che vuole sapere a runtime la cardinalità di tale enumerazione, non esiste un metodo Factory perché non c'è motivo di riutilizzare classi oppure fornire un implementazione diversa, siccome prende una chiave ed un valore è implementata tramite array indipendentemente dalla grandezza e funziona a tempo costante.

17.4 Esercizio subMap

Il metodo subMap accetta una mappa ed una collezione, restituisce una nuova mappa ottenuta restringendo la prima alle sole chiavi che compaiono nella collezione, il metodo non modifica i parametri in ingresso.
Abbiamo scelto il contratto che accetta qualsiasi tipo di mappa e collezione.

1. `<K> Map<K, ?> subMap(Map<K, ?> m, Collection<K> c);`
   Analisi intestazione: Funzionale, non completa (Richiede che la collezione e la chiave coincidano), corretta, offre discrete garanzie su m.
   Analisi ritorno: Troppo generico, l'oggetto è poco utile al chiamante, presa una mappa concreta ne ottiene una generica.
2. `<K, V> Map<K, V> subMap(Map<K, V> m, Collection<?> c);`
   Analisi intestazione: Funzionale, completa, corretta, offre forti garanzie su c, meno semplice della precedente (ma è completa, cosa più importante).
   Analisi ritorno: Tipo di ritorno sufficientemente specifico, ritorna una mappa del tipo corretto e non è troppo specifico su come è implementata.
3. `<K, V> Map<K, V> subMap(Map<K, V> m, Collection<? super K> c);`
   Analisi intestazione: Funzionale, non completa perché `<? super K>` non accetta qualsiasi Collection, corretta, offre parziali garanzie su c, equivalente alla 2 in semplicità.
   Analisi ritorno: Come la 2.
4. `<K, V, K2 extends K> Map<K, V> subMap(Map<K, V> m, Collection<K2> c);`
   Analisi intestazione: Funzionale, non completa perché scarta le cose che non siano sottotipo di K, corretta, non offre garanzie su c, inoltre è la più complessa per il numero di parametri.
   Analisi ritorno: Come la 2.

La conclusione è che la migliore è la 2, analizzando se è possibile migliorarla potremmo togliere `V` con `?` ma perderemmo il vantaggio del tipo di ritorno, quindi non è una buona idea, però possiamo aumentare le garanzie sulla mappa come segue:

```
<K,V> Map<K,V> subMap(Map<? extends K, ? extends V> m, Collection<?> c) {
  HashMap<K, V> results = new ...
  for(K key : m.keySet()) {
     if(c.contains(key)) {
        results.put(key, m.get(key));
     }
  }
}
```

Possiamo fare `<? extends K, ? extends V>` in quanto nel corpo non ci serve il tipo specifico di questi e ne guadagniamo in garanzie.

17.5 Threads

Sono una forma di programmazione concorrente in cui il programma si "divide" in vari flussi di esecuzione e questi suoi flussi non sono indipendenti, i Thread rispetto ai processi condividono la memoria e quindi possono leggere e scrivere su variabili create da altri thread, i thread tra di loro hanno solo stack e program counter diverso.

Avere memoria condivisa è un arma a doppio taglio, velocizza la comunicazione tra di essi ma possono crearsi *race conditions* che creano seccature per la sincronizzazione.

Molti linguaggi di programmazione supportano nativamente il multithread a differenza del `C` che richiede una libreria esterna per farlo ed utilizza *system call*, tra i tanti linguaggi Java è stato il primo a supportare il multithread nativamente.

Il supporto è basato sulla classe `Thread`, ipotizziamo di voler scrivere un programma che crea un secondo thread (il primo thread è sempre il main) e gli fa stampare qualcosa:

```
main(...) {
  Thread t = new Thread() {
     @override
     public void run() {
        System.out.println("Hello");
     }
  }
  t.start();
  System.out.println("world");
}
```

Si crea una sottoclasse di Thread, in questo caso sotto forma di classe anonima, si fa l'override del metodo `run()` che corrisponde all'entry point del thread, lì dentro va scritto il codice che si intende eseguire in parallelo, la creazione della classe però non fa partire il thread, per quello è necessario invocare il metodo `start()` che restituisce immediatamente il controllo al chiamante mentre fa partire il thread.

Si noti che il metodo `start()` può essere chiamato **una sola volta** ed inoltre **non ci sono**

garanzie su quale stampa avverrà prima, se finirà prima il main oppure il nuovo thread, ed altro sull'ordine di essi, di fatto eseguendo più volte lo stesso codice si avranno comportamenti diversi in base al dispatcher del sistema operativo.

Un Thread si può costruire utilizzando anche un altro costruttore (oltre quello vuoto, usato prima) che prende al suo interno un oggetto `Runnable`, quest'ultima è un interfaccia che rappresenta l'entry point del thread, la sua interfaccia:

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

Il vantaggio di questo costruttore è di non dover creare una classe anonima, vediamo lo stesso esempio precedente ma con il nuovo costruttore:

```
main(...) {
  Thread t = new Thread(new Runnable {
     @override
     public void run() {
        System.out.println("Hello");
     }
  };)
  t.start();
  System.out.println("world");
}
```

Ulteriore vantaggio è che Runnable è un interfaccia funzionale, quindi possiamo sostituirlo con una lambda espressione:

```
main(...) {
  Thread t = new Thread( () -> System.out.println("Hello"));
  t.start();
  System.out.println("world");
}
```

Nonostante sia un grande vantaggio è bizzarro vederlo racchiuso in una lambda espressione oltre che a scopo didattico, un thread dovrebbe essere creato per eseguire un blocco di istruzioni e non cose così semplici.

Un thread inizia con il metodo `start()` e termina al `return` del suo metodo `run()`

Vediamo vari metodi della classe `Thread`, molti dei quali appena utilizzati:

```
void run();
void start();
Thread();
Thread(Runnable r);
static Thread currentThread(); // Contratto: Restituisce l'oggetto Thread
    corrispondente al thread di esecuzione di chi lo chiama
```

18 Lezione 18

La lezione 18 continua sui thread, il loro scopo, l'esercizio delayIterator, la disciplina delle interruzioni e l'esercizio Cardinal.

18.1 Continuo sui Threads

18.1.1 Scopo del multi-Threading

I motivi per cui scegliere di programmare in parallelo sono i seguenti:

1. Banalmente avere più blocchi di codice eseguiti contemporaneamente, massimizzando l'utilizzo della cpu senza sprecarla
2. Su hardware single core (Che quindi non supporta nativamente il multi-threading) aiuta a separare i compiti che non hanno molta comunicazione tra di loro
3. Massimizzare l'utilizzo di tutte le risorse, ovvero presa un'operazione lenta come quelle di I/O, ad esempio il salvataggio dei dati, delegarla ad un thread a parte rende possibile sfruttare la cpu mentre altrove viene salvata l'informazione

18.1.2 Esercizio DelayIterator

Abbiamo visto la classe Thread e l'interfaccia Runnable, possiamo applicare queste due novità nell'esercizio DelayIterator di Marzo 2008, si consideri il seguente caso d'uso:

```
List<Integer> myList = new ArrayList<>();
myList.add(...);
Iterator<Integer> i = delayIterator(myList.iterator(), 2);
while(i.hasNext()) {
  System.out.println(i.next());
}
```

Vogliamo che il metodo ritorni un iteratore che si ferma per 2 secondi tra una stampa e l'altra, iniziamo a vedere che firma può avere questo metodo:

```
public static <T> Iterator<T> delayIterator(Iterator<T> i, int delay);
```

Seguendo il caso d'uso dichiariamo il metodo parametrico per preservare il tipo dell'iteratore, questa firma però può essere migliorata dando ulteriori garanzie all'iteratore che verrà unicamente letto, procediamo con l'implementazione:

```
public static <T> Iterator<T> delayIterator(Iterator<? extends T> i, int delay) {
  // Classe anonima
  return new Iterator<T>() {
     @override
     public boolean hasNext() {
        // i e' effectively final e posso usarla
        return i.hasNext();
     }

     @override
```

```
11      public T next() {
12         try {
13            // Puo' lanciare eccezioni verificate
14            Thread.sleep(delay * 1000);
15         }
16         catch(InterruptedException e) {}
17         return i.next();
18      }
19   }
20 }
```

Nella riga 14 richiamiamo il metodo `sleep` che è il primo tra i 3 metodi "bloccanti", questo metodo prende un tempo di attesa in millisecondi e lancia l'eccezione verificata `InterruptedException` nel caso in cui ne venga interrotta la pausa, di conseguenza il compilatore ci obbliga e gestirla con un **try-catch**.

Quando si verifica la catch abbiamo 2 scelte:

1. Lanciarne una non verificata, se riteniamo che l'attesa di 2 secondi tra una print e l'altra sia essenziale
2. Restituire l'elemento anticipatamente, se riteniamo che l'attesa non sia fondamentale, ignorando l'eccezione

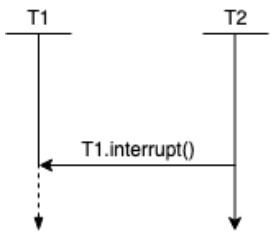
Abbiamo scelto la seconda opzione, però si noti che lasciare un blocco `catch` vuoto è molto spesso un *Code Smell* ed andrebbe evitato.

18.1.3 Disciplina delle interruzioni

Per interrompere un thread si utilizza un metodo istanza della classe Thread su un altro oggetto Thread:

```
1 public void interrupt();
```

Può sembrare uno scioglilingua, per chiarire meglio si veda la seguente immagine con T1 e T2 due Thread:



Richiamando T1.interrupt() viene impostato il flag di stato a false di T1 e non cambia lo stato di T2, si noti che questo non termina il thread in quanto T1 deve "concordare" la sua terminazione, per questo viene chiamata *Disciplina delle interruzioni*, questa prevede che due thread collaborino per la terminazione.

Non esiste un metodo che termini in modo forzato un thread in quanto T1 in quel momento potrebbe avere file aperti, usare mutex o altre risorse di sistema, terminare forzatamente il thread

in quello stato potrebbe farlo entrare in una situazione dove le risorse non sarebbero liberate e risulterebbero sempre occupate.

Affinché T1 rispetti questa disciplina è necessario aggiungere un metodo che serve ad interrogarsi sul proprio stato, questo metodo è utile solo se T1 non ha chiamate bloccanti, altrimenti è superfluo:

```
public boolean isInterrupted();
```

18.1.4 Esercizio Interruptor (Terminazione di thread)

Prendiamo l'esercizio Interruptor, il cui compito è quello di creare un nuovo thread che interrompe quello in ingresso dopo un n numero di secondi e viene costruito nel seguente modo:

```
Interruptor i = new Interruptor(t, 10);
```

Procediamo con l'implementazione, creiamo un nuovo thread perché vogliamo che questo costruttore non sia bloccante:

```
class Interruptor {
  public Interruptor(final Thread t, final int delay) {
     Thread thread = new Thread() {
        public void run() {
           try {
              Thread.sleep(delay*1000);
           }
           catch(InterruptedException e) {
              return;
           }
           t.interrupt();
        }
     };
     thread.start();
  }
}
```

Il costruttore crea il thread e lo avvia, siccome questo è un nuovo thread deve interpretare il catch dell'eccezione come una richiesta di terminazione, però per completarlo dobbiamo aggiungere concretamente un metodo che ci permetta di farlo, quindi dobbiamo togliere la classe anonima e rendere il thread come un oggetto della classe:

```
class Interruptor {
private Thread thread;

  public Interruptor(final Thread t, final int delay) {
     thread = new Thread() {
        public void run() {
           try {
              Thread.sleep(delay*1000);
           }
           catch(InterruptedException e) {
```

```
            return;
          }
          t.interrupt();
        }
      };
      thread.start();
    }

    public void cancel() {
      thread.interrupt();
    }
}
```

18.1.5 Esercizio consumeSet (metodo join)

Prendiamo l'interfaccia funzionale `Consumer` già vista nel capitolo delle lambda espressioni, questa interfaccia prende un oggetto e non restituisce nulla (quindi lo "consuma"):

```
Consumer<T> {
  void accept(T t);
}
```

Creiamo un metodo `consumeSet` che esegue il consumatore su tutti gli oggetti del Set contemporaneamente, inoltre vogliamo che sia un metodo bloccante che restituisce il controllo solo quando tutti gli oggetti sono stati consumati, iniziamo con la possibile firma:

```
public static <T> void consumeSet(Set<T> set, Consumer<T> c);
```

Può essere migliorata? Per farlo ci serve stabilire un contratto, ipotizziamone uno che non chiede esplicitamente 2 cose dello stesso tipo ma accetta anche sottoclassi, in quel caso non è completa, quindi rendiamola completa seguendo quest'ultimo:

```
public static <T> void consumeSet(Set<T> set, Consumer<? super T> c);
```

Le linee guida Java consigliano allora di cambiare anche Set come segue per ulteriori garanzie:

```
public static <T> void consumeSet(Set<? extends T> set, Consumer<? super T> c);
```

Usiamo `extends` per leggere e `super` per scrivere. Dopo questa analisi possiamo procedere alla scrittura del metodo:

```
public static <T> void consumeSet(Set<T> set, Consumer<? super T> c) throws
    InterruptedException {
  // Siccome so la grandezza la sfrutto per aumentare le performance di ArrayList
  List<Thread> threads = new ArrayList<>(set.size());

  /* Creiamo i thread i vari thread e li facciamo partire, teoricamente non partono
     tutti nello stesso istante ma hanno un 1ns di differenza, si noti che non e'
     possibile fare meglio di cosi' */
  for(T t : set) {
```

```
    Thread x = new Thread(() -> c.accept(t));
    x.start();
    threads.add(x);
  }

  for(Thread x : threads) {
    x.join();
  }
}
```

Il metodo `join()` è il secondo metodo "bloccante" e mette il thread corrente in attesa della terminazione di `this` thread, questo metodo ha a che fare con il thread chiamante ed il thread su cui è chiamato, in questo caso x, se si chiama su un thread già terminato restituisce all'istante il controllo al chiamante.

Siccome `join()` è bloccante (lancia `InterruptedException`) allora anche consumeSet è un metodo bloccante e segnaliamo il lancio dell'eccezione nella firma.

18.2 Esercizio Cardinal (enum)

```
enum Cardinal {
  // Si noti che e' importante l'ordinamento
  N, NNE, NE, ENE, ..., S, ..., W, ...;

  public boolean isOpposite(Cardinal e) {
    return (((this.ordinal() + 8) % 16) == e.ordinal());
  }

  public static Cardinal max(Cardinal e, Cardinal e2) {
    if(e.isOpposite(e2)) {
      throw new IllegalArgumentException();
    }

    int res = (e.ordinal() + e2.ordinal()) / 2;

    if(math.abs((e.ordinal() + e2.ordinal()) > 8) {
      return Cardinal.values()[res + 8 % 16];
    }
    return Cardinal.values()[res];
  }
}
```

19 Lezione 19

"Non vorrei parlare sempre di linguaggi di programmazione in un corso di linguaggi di programmazione"
- M. Faella

La lezione si apre con due libri, non inerenti al corso, consigliati dal docente, il primo è *Managing Oneself* di Peter Drucker, un brevissimo libro sull'autogestione e con spunti un po' filosofici.
Il secondo libro è *Teaching and learning STEM*, un libro più compatto e pratico rivolto agli insegnanti, distingue molti modi per imparare.

La lezione 19 parla dei Thread sincroni, le classi Thread-Safe, le Condition Variables ed il paradigma Consumatori-Produttori.

19.1 Thread sincroni (synchronized)

Abbiamo visto le funzionalità base del multithreading ma con forti limitazioni sull'interazione tra di loro, in questo capitolo vedremo le problematiche e le soluzioni con la sincronizzazione tra thread.

Partiamo con un esempio, abbiamo un determinato task ed un incremento per contare quante volte questo task viene eseguito:

```
1  class Task implements Runnable {
2    public void run() {
3       ...
4       counter++;
5    }
6  }
7
8  main(...) {
9    Task task = new Task();
10   Thread t1 = new Thread(task), t2 = new Thread(task), t3 = new Thread(task);
11
12   t1.start();
13   t2.start();
14   t3.start();
15 }
```

A fine programma vorremmo trovarci come valore di counter 3, la prima domanda che sorge è dove mettere la dichiarazione di counter? Non possiamo dichiararla locale di `run()` perché altrimenti ogni thread avrebbe un counter separato, di conseguenza la dichiariamo come variabile istanza della classe Task, si ipotizzi dichiarata come `private int counter`

Scopriamo ora i vari problemi di *race condition*, questo codice compila ma a fine programma potrei non trovarmi 3, questo avviene per il semplice motivo che l'incremento non è un operazione atomica, ovvero mentre avviene un incremento altri thread potrebbero compiere altre azioni su quella stessa variabile, questo rende i vari possibili output del programma 1, 2 o 3.

Ci sono alcune primitive in Java che consentono di creare **mutex** (mutua esclusione), queste sono anche dette *sezioni critiche*, in Java il meccanismo base è un semaforo binario (libero, occupato) ed

ha due metodi: `lock()` e `unlock()`, questi mutex non hanno una loro classe ma sono mascherati e gestiti dalla keyword `synchronized`.

Ad ogni oggetto in Java è automaticamente associato un mutex, questo viene chiamato *"monitor"* di quell'oggetto, si noti che i tipi primitivi non hanno mutex associato in quanto non sono oggetti, ogni oggetto ha questo piccolo overhead di memoria per contenere questi particolari sistemi di mutex, siccome sono contenuti in ogni oggetto non è necessario crearli esplicitamente.

Un blocco `synchronized`, che viene usato per bloccare e rilasciare risorse, ha la seguente sintassi:

```
synchronized(exp) {
  // Blocco istruzioni
}
```

Di cui si noti che:

1. `exp` deve essere un oggetto oppure espressione che ha come risultato un oggetto
2. { blocca il monitor di quell'oggetto
3. } libera il monitor di quell'oggetto

La keyword `synchronized` può essere anche utilizzata prima di un metodo, in quel caso il metodo attende che la variabile di exp sia liberata prima di partire.

Nota importante una variabile non può essere dichiarata come `synchronized`, questa keyword si può utilizzare solo su metodi o per blocchi di codice.

Si noti che la `synchronized` funziona anche sui metodi statici ma con un comportamento diverso, si veda il seguente esempio:

```
class A {
  synchronized static f() {}
}

// Esattamente equivalente a scrivere
class A {
  f() {
     synchronized(A.class) {
        ...
     }
  }
}
```

Dopo aver mostrato queste regole facciamo in modo che a fine programma il risultato di counter sia sempre 3 applicando `synchronized`

```
class Task implements Runnable {
  private int counter;
  public void run() {
     ...
     synchronized(this) {
        counter++;
     }
```

```
  }
}
```

Abbiamo usato `synchronized(this)` in quanto la classe task è condivisa tra i vari thread, analizziamo altre alternative:

1. `synchronized(counter)`, Errore di compilazione, counter è un tipo primitivo
2. Se counter fosse di tipo Integer, avremmo potuto provare `synchronized(counter)`, ma questo ha un altro problema, compila però ha la seccatura che counter++ cambia counter in quanto gli Integer sono immutabili, creando un nuovo oggetto, questo provoca problemi più grandi di quanto si possa pensare, si noti il seguente esempio:

2.1. T1 entra in synchronized con counter a 0 e fa counter++, counter adesso punta ad un nuovo oggetto Integer a cui T1 punta con un nuovo mutex.

2.2. Se T2 stava dormendo fin qui allora siamo fortunati e funziona, ma ipotizziamo che T2 era attivo e si ferma appena entrato in synchronized(0) perché attende T1.

2.3. T3 parte e T1 finisce, permettendogli di entrare subito nel blocco, questo viola la mutua esclusione.

3. Lasciare `synchronized(this)` anche con counter Integer, funziona correttamente
4. Rendere il metodo `run` sincronizzato come segue: `public synchronized void run()`, la domanda che potrebbe sorgere è se questo sia un overriding valido, questo non è un problema in quanto è sempre lecito fare overriding sincronizzati di metodi non sincronizzati e viceversa. Questo è consentito in quanto la sincronizzazione è un dettaglio implementativo, possiamo affermare che la keyword non altera il contratto del metodo. Infine come affermato precedentemente `synchronized` su un metodo è equivalente a `synchronized(this)` e funziona correttamente.

Modifichiamo adesso il main, a differenza dell'esempio precedente ipotizziamo che ogni thread abbia un task diverso:

```
main(...) {
  Thread t1 = new Thread(new task(...)),
  t2 = new Thread(new task(...)),
  t3 = new Thread(new task(...));

  t1.start();
  t2.start();
  t3.start();
}
```

In questo caso ci tocca modificare la variabile di task e renderla statica:

```
class Task implements Runnable {
  private static int counter;
  public void run() {
     ...
     synchronized(this) {
        counter++;
```

```
    }
  }
}
```

Però avendo tutti task diversi non funziona più `synchronized(this)` e non rimuove la race condition, per risolvere questo possiamo utilizzare l'oggetto Class di task, siccome voglio che tutti competano per questo oggetto:

```
class Task implements Runnable {
  private static int counter;
  public void run() {
     ...
     synchronized(Task.class) {
        counter++;
     }
  }
}
```

Aggiungiamo un ulteriore variazione:

```
class Task implements Runnable {
  private static Object lock = new Object();
  private static int counter;
  public void run() {
     ...
     synchronized(lock) {
        counter++;
     }
  }
}
```

Questo altro metodo funziona correttamente ed è anche la **best-practice** e la soluzione migliore rispetto a tutti i casi precedenti, sincronizzare su un oggetto privato piuttosto che passare l'oggetto Class, questo evita alcuni problemi che potrebbero sorgere se si volesse accedere alla classe Task da altrove, in quanto la troverebbero bloccata dal metodo che utilizza Task.class

Teoricamente un'altra soluzione era utilizzare AtomicInteger (una classe thread safe) che supporta l'incremento in modo atomico, permettendo la sincronizzazione in maniera implicita, ma non è argomento del corso.

19.2 Classi thread safe

In C un metodo si dice thread safe quando garantisce che più thread che lo chiamano concorrentemente non causano effetti indesiderati, più formalmente diciamo che una classe è thread-safe se: *"La funzione rispetta il suo contratto anche quando invocata concorrentemente da più thread"*

Una classe thread safe ha lo stesso principio, ovviamente esteso per considerare anche i campi, tutte le collezioni studiate finora non sono thread safe, questa è una scelta implementativa in quanto questo le rende più efficienti.

Supponiamo la seguente situazione:

```
class A {
  synchronized void f();
  synchronized void g();
  void h();
}

main(...) {
  A a = ..., b = ...;
  // T1 invoca a.f();
}
```

Che succede se T2 invoca:

- `a.f()`? Viene sospeso finché T1 non termina
- `b.f()`? Viene eseguito
- `a.g()`? Viene sospeso finché T1 non termina
- `a.h()`? Viene eseguito

Supponiamo ora la seguente modifica ad A:

```
class A {
  synchronized void f() { g(); };
  synchronized void g();
  void h();
}
```

Abbiamo che un thread può ottenere più volte lo stesso mutex, di conseguenza distinguiamo *mutex rientranti e non rientranti*, il primo caso permette allo stesso thread di ottenere nuovamente il mutex, questi hanno bisogno di un contatore (vanno liberati lo stesso numero di volte che vengono presi) ed in Java tutti i mutex sono rientranti.

19.3 Condition variables

Le condition variables risolvono il problema dell'attesa di un evento arbitrario, si ipotizzi una situazione in cui un secondo thread deve attendere il completamento del primo per avanzare, in maniera ingenua si potrebbe fare come segue:

```
boolean done = false;

// Thread 1
// Fa qualcosa
done  = true;

// Thread 2, in attesa con un ciclo che non fa nulla
while(!done) {}
```

Qui abbiamo che il secondo thread è in attesa attiva, questo è il 95% delle volte un errore logico in quanto può essere risolto senza attesa attiva, scriviamolo con una condition variable:

```java
boolean done = false;
Object lock = new Object();

// Thread 1
// Fa qualcosa
synchronize(lock) {
  done = true;
  lock.notifyAll(); // Risveglia tutti i thread in wait
}

// Thread 2
while(!done) {
  synchronize(lock) {
     try {
        lock.wait(); // Mette il thread in attesa di un notify/notifyAll
     }
     catch(InterruptedException e) {
        return;
     }
  }
}
```

Il metodo `notifyAll` è un metodo di Object, ad ogni oggetto oltre un monitor viene associata una *condition variable*, ovvero una lista di condition variables in attesa, il metodo `wait()` si occupa di mettere quella variabile in attesa e si ferma in attesa di un `notifyAll()`, wait pretende il possesso del monitor, ecco perché viene messo in un blocco sincronizzato, inoltre essendo una chiamata bloccante può lanciare `InterruptedException`.

Mettere la wait in un while può sembrare una mossa strana, ma questo è necessario per gli *"spurious wake-up"*, in quanto se il thread dovesse svegliarsi senza motivo verrebbe subito messo di nuovo in wait in quanto `done` risulta ancora falsa.

19.3.1 Contratto di wait e notifyAll

Il contratto di `x.wait()` è il seguente:

1. Se il thread corrente non ha il monitor di x, lancia eccezione
2. In un'unica operazione atomica:

2.1. Inserisce il this corrente nella lista di attesa di x
2.2. Rilascia il monitor di x
2.3. Sospende l'esecuzione del thread corrente

3. Dopo una notify (dove è stato scelto questo thread) o una notifyAll, riprende il monitor e restituisce il controllo al chiamante
4. Se lo stato di interruzione diventa true, lancia `InterruptedException`
5. In casi eccezionali la wait può risvegliarsi anche senza notify, questi vengono chiamati *"spurious wake-up"*

Il contratto di `x.notifyAll()` è il seguente:

1. Se il thread corrente non ha il monitor di x, lancia eccezione
2. Risveglia tutti i thread nella lista di attesa di x

19.4 Introduzione al paradigma produttori-consumatori

È una situazione ricorrente della programmazione multithread, si tradda di due o più thread divisi in due categorie:

- I produttori, sono fonti di informazioni destinate ai consumatori
- I consumatori, devono elaborare le informazioni fornite dai produttori, appena disponibili

Non è possibile prevedere quanto tempo impiega un produttore per produrre informazione, né quanto impiega un consumatore ad elaborarla, questo genera il problema della sincronizzazione delle operazioni tra i thread.

La soluzione classica prevedere uno o più *buffer* che contengono le informazioni prodotte e non ancora consumate, nella nostra ipotesi questo buffer è unico e con capienza limitata, se un produttore lo trova pieno attende che venga liberato un posto, viceversa se un consumatore lo trova vuoto attende che un produttore inserisca informazione, per fare questo utilizzeremo una condition variable evitando race condition.

Si ipotizzi che `buf` punti ad una struttura dati che con put inserisce un elemento e con take lo rimuove, questi thread utilizzeranno questo buffer non solo per comunicare ma anche 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);
  buf.notifyAll() // Notifica i consumatori
}

// Consumatore
synchronized(buf) {
  // Attende che il buffer non sia vuoto
  while(buf.isEmpty()) {
     try {
        buf.wait()
     } catch (InterruptedException e) {
        return;
     }
  }
```

```
25    some_value = buf.take();
26    buf.notifyAll(); // Notifica i produttori
27  }
```

Questo schema è adatto anche alla situazione con tanti produttori e tanti consumatori, viene naturale chiedersi perché sia il produttore che il consumatore basino la propria attesa su un ciclo `while` rispetto ad un semplice `if`.

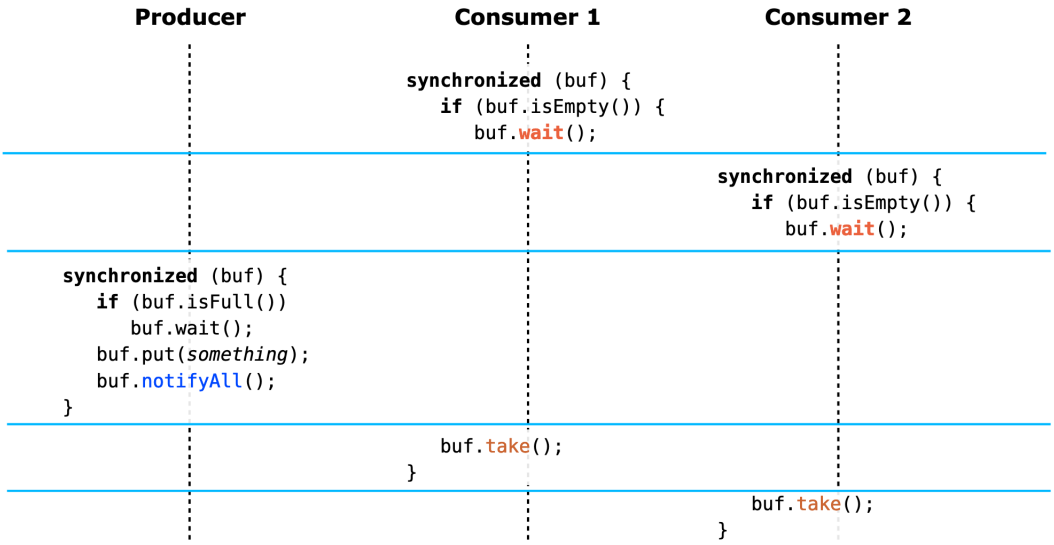
19.4.1 Problemi del paradigma

Ipotizziamo di sostituire il while come segue:

```
1  if(buf.isEmpty()) {
2    buf.wait();
3  }
4  ...
```

Si possono verificare tre problemi:

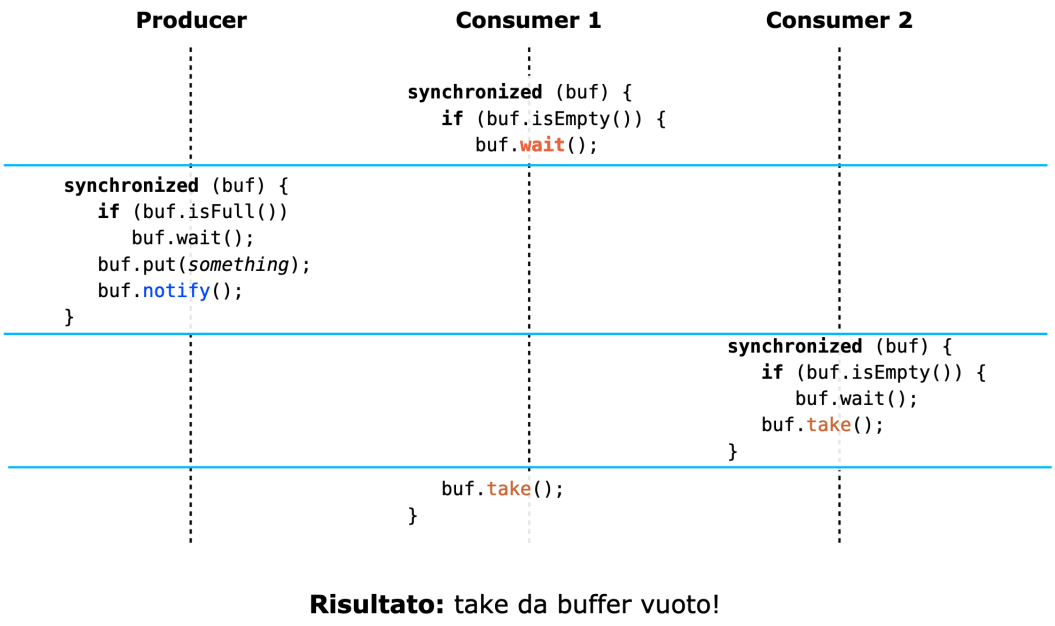
1. Se il produttore usa notifyAll, vengono svegliati due consumatori ma c'è un solo valore nel buffer



Risultato: take da buffer vuoto!

   Qui si verifica la situazione in cui partono entrambi i consumatori e si mettono in attesa trovando il buffer vuoto, poi un produttore produce e li risveglia, ma siccome entrambi avevano passato il controllo precedente il consumatore 1 prende l'elemento ed il consumatore 2 non lo trova.

2. Se il produttore usa notify, viene svegliato un solo consumatore ma un altro lo anticipa

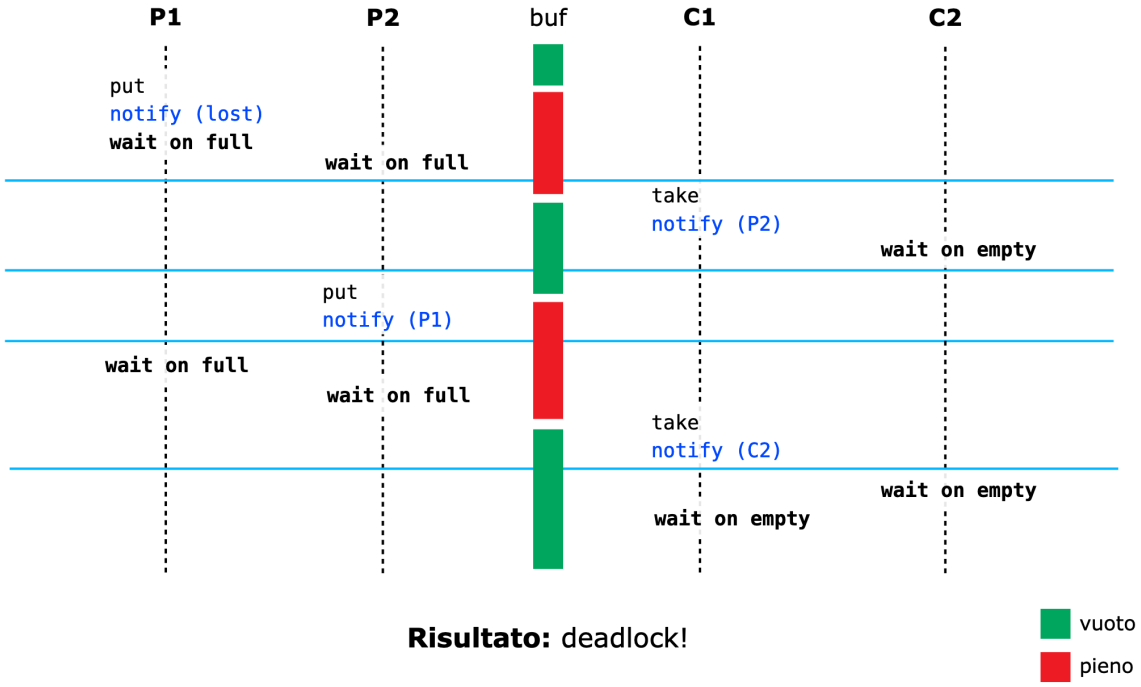


Risultato: take da buffer vuoto!

In questo caso abbiamo che il primo consumatore si mette in attesa, poi il produttore produce un elemento, però prima che possa lanciare la notify si verifica che un secondo consumatore parte e trova il buffer pieno, quindi lo prende prima del risveglio del primo consumatore, che lo trova vuoto

3. In tutti i casi, uno spurious wake up da wait può portare a leggere da un buffer vuoto

Si noti però che la notifyAll conviene anche se ogni prodotto è destinato ad un unico consumatore, usare una singola notify può creare un deadlock, si ipotizzi di avere un buffer di capienza 1, condiviso tra due produttori e due consumatori:



Risultato: deadlock!

Quello che accade è che parte il primo produttore, crea e lancia una notify che però non sveglia nessuno essendo l'unico thread, poi parte il secondo produttore e parte il primo consumatore, che consuma e notifica il secondo produttore, nel frattempo è partito il secondo consumatore in attesa.
Il secondo produttore parte, produce e risveglia P1, che va subito in attesa, poi parte C1 che prende un elemento e risveglia C2 che va subito in attesa, ed ecco che ci troviamo in deadlock per una serie di risvegli "sventurati".

20 Lezione 20

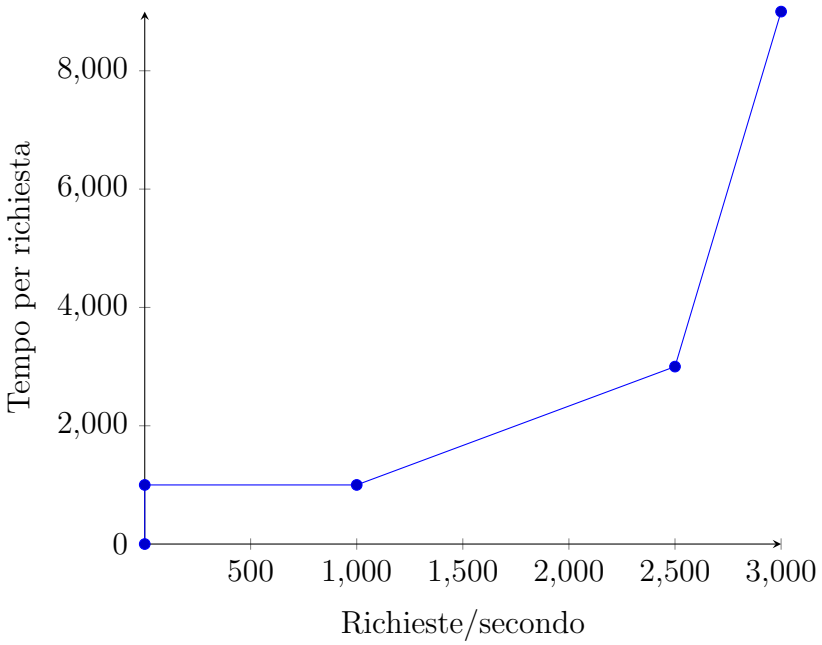
"Avete votato? Avete votato BENE? Perché non mi sembra"
- M. Faella

La lezione 20 approfondisce il paradigma produttori-consumatori, introduce le code bloccanti ed il Java Memory Model.

20.1 Approfondimento sul paradigma produttori-consumatori

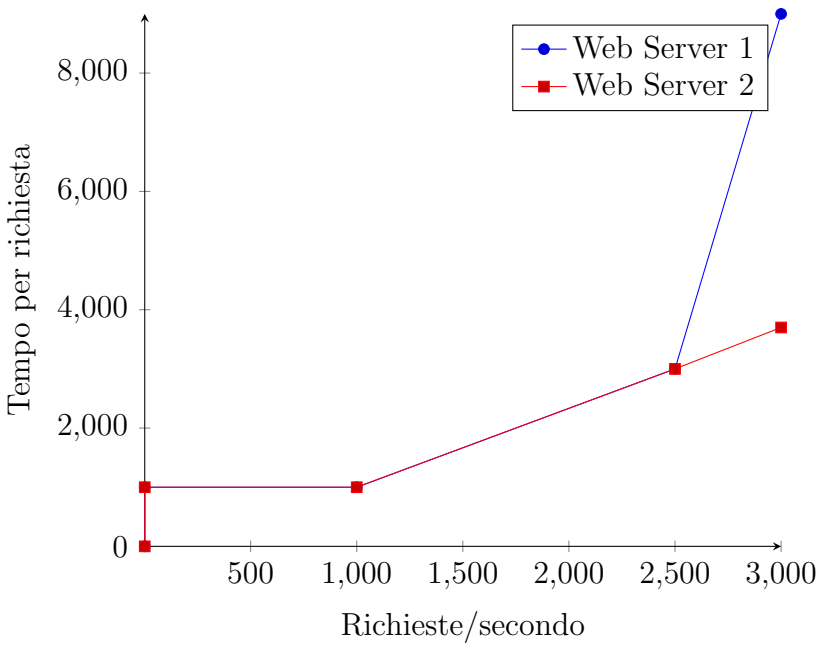
Tutti i casi particolari precedentemente mostrati servivano a giustificare il fatto che tutti devono svegliare tutti tramite `notifyAll` e che tutti devono ricontrollare la propria condizione di attesa con un while, questa è l'unica soluzione esente da deadlock e race condition se si vuole usare un unica condition variable, se tutti usano quest'unica condition variable è la migliore, esistono anche evoluzioni che prevedono più condition variables che ne possono aumentare l'efficienza e ridurre risvegli inutili di thread.

Vediamo un applicazione concreta di questo paradigma, supponiamo di avere un web server con un architettura con un unico processo che apre un socket di ascolto ed ogni nuova connessione apre un nuovo thread per rispondere a tale richiesta. Ipotizziamo inoltre di utilizzare un computer di fascia media, il suo andamento nel tempo sarà il seguente:



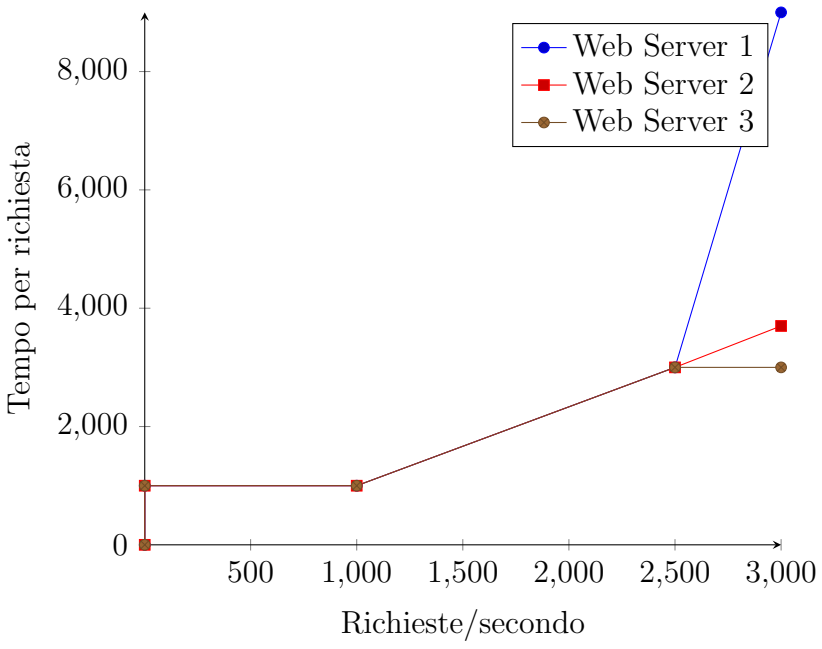
Si noti che il tempo impiegato non è subito crescente in quanto *"Non è mica un commodore 64"*, quindi per la prima fase è ottimo, il serve per le prime 1000 richieste rispetta il tempo medio, dopodiché crea più thread di quanti ne terminano ed i tempi iniziano a salire, fino ad un certo punto dove il sistema collassa in quanto è impegnato a creare solo thread senza riuscire a soddisfare alcuna richiesta.

Ipotizziamo ora un architettura più elaborata, il server mette in un buffer le richieste e poi c'è un numero fissato di consumatori di richieste, ipotizziamo K thread con buffer illimitato, questo K sarà stabilito dal programmatore in base a quello che dovrà compiere il sistema, ad esempio se le operazioni sono legate a calcoli allorà $K = cores$ altrimenti potrebbe essere anche $K = cores*2$, vediamo l'andamento di questo sistema:



Come per il server precedente, per le prime richieste è costante, poi dovrà attendere la liberazione di thread, la differenza risiede quando si supera il punto di collassamento che è migliore in quanto continua a mettere i vari thread in coda, anche se migliore arriverà ad un punto, anche con buffer illimitato, in cui il sistema collasserà.

Come ultima ipotesi, prendiamo il Web server 2 con un buffer limitato, e più realistico, essendo limitato quando il buffer termina allora viene bloccata la richiesta e da un tempo massimo che si può impiegare per soddisfarle.



Il migliore tra queste ipotesi è il server 3 in quanto il server 2 degrada il tempo del servizio a tutti man mano, il server 3 assicura sempre un tempo massimo per ottenere risposta, quindi riassumendo:

1. WebServer 1

1.1. Ogni richiesta genera un nuovo thread
1.2. Ad un certo punto si verifica un collassamento del sistema

2. WebServer 2

2.1. K thread consumatori con buffer illimitato
2.2. Degrada il tempo medio man mano che aumentano richieste

3. WebServer 2 con buffer limitato

20.2 Queue bloccanti

Aggiungiamo qualcosa alle collezioni già viste, l'interfaccia Queue ha vari metodi, tra cui `peek()`, `put()` e `take()`, si noti che `LinkedList<T>` implementa sia List che Queue, ma la sottointerfaccia più interessante è `BlockingQueue<T>`.

Le due implementazioni di `BlockingQueue` sono `ArrayBlockingQueue` e `LinkedBlockingQueue` che sono classi thread safe con opportuna sincronizzazione, inoltre sono entrambe FIFO in quanto put mette in coda e take prende in testa.

Diamo un rapido sguardo ai contratti dei tre metodi sopracitati:

- `peek()`: Restituisce il primo elemento della coda senza modificarla
- `put()`: Inserisce in coda ed è bloccante se la coda è piena, ovvero rimane in attesa di inserire
- `take()`: Rimuove un elemento dalla coda ed è bloccante se la coda è vuota

Gli ultimi due metodi rispettano la disciplina delle interruzioni e sono sensibili allo stato di interruzione del thread corrente, possono sollevare entrambi `InterruptedException`.

`ArrayBlockingQueue` come costruttore prende la grandezza dell'array e non viene ridimensionato.

20.2.1 Esercizio ThreadRace (join, notifyAll e BlockingQueue)

Vogliamo scrivere un metodo statico ThreadRace che accetta 2 runnable, li esegue in parallelo e restituisce quello che finisce per primo, una cosa non specificata nell'esercizio è se restituisce il controllo quando terminano entrambi o solo uno dei due, vediamo prima il primo caso:

```
1  static int threadRace(Runnable a, Runnable b) {
2    int[] result = new int[1];
3
4    class MyThread extends Thread throws InterruptedException {
5       int id;
6       Runnable r;
7       MyThread(Runnable r, int id) { /*solito costruttore*/ }
8
9       public void run() {
10         // Richiamo il run del runnable associato a questa classe
11         r.run();
12
```

```
13          synchronized(result) {
14             // Se e' 0 nessuno ha ancora scritto
15             if(result[0] == 0) {
16                result[0] = id;
17             }
18          }
19       }
20    }
21
22    Thread t1 = new MyThread(a, 1), t2 = new MyThread(b, 2);
23
24    t1.start();
25    t2.start();
26
27    // Aspetto che entrambi terminino
28    t1.join();
29    t2.join();
30
31    // Qui non ho race condition perche' t1 e t2 saranno terminati
32    return result[0];
33 }
```

Si noti una serie di scelte implementative:

1. Nella riga 2 creiamo un array con un singolo elemento, questo è necessario in quanto non possiamo usare un int perché il metodo della classe interna `run(()` vuole qualcosa che sia effectively final, però dichiasrandolo come array è possibile modificarlo (il riferimento è final, non il contenuto), la soluzione migliore ancora una volta sarebbe AtomicInteger ma è fuori dal programma
2. Nella riga 4 dichiaro una nuova classe interna, questo è dovuto al fatto che se avessi voluto fare 2 classi anonime avrei duplicato il codice
3. Nella riga 13 prima di scrivere mettiamo un blocco synchronized in quanto la variabile è condivisa tra i thread, inoltre possiamo farlo perché gli array sono oggetti in Java
4. Nelle righe 28 e 29, sebbene `join()` sia un metodo bloccante non mettiamo il try e catch bensì lo scriviamo nella firma del metodo, abbiamo deciso che il metodo sia bloccante ed è responsabilità del chiamante occuparsi dell'eccezione

Facciamo ora la variante in cui quando termina il più veloce dei due termina senza aspettare l'altro più lento, per questo non possiamo più usare la join ma dobbiamo usare wait e notify.

```
1  static int threadRace(Runnable a, Runnable b) {
2     int[] result = new int[1];
3
4     class MyThread extends Thread throws InterruptedException {
5        int id;
6        Runnable r;
7        MyThread(Runnable r, int id) { /*solito costruttore*/ }
8
9        public void run() {
```

```
        // Richiamo il run del runnable associato a questa classe
        r.run();

        synchronized(result) {
           // Se e' 0 nessuno ha ancora scritto
           if(result[0] == 0) {
              result[0] = id;
              result.notifyAll();
           }
        }
     }
  }

  Thread t1 = new MyThread(a, 1), t2 = new MyThread(b, 2);

  t1.start();
  t2.start();

  // Aspetto che termini almeno uno
  synchronized(result) {
     while(result[0] == 0) {
        result.wait();
     }
     return result[0];
  }
}
```

Come già visto, il while su result ci protegge da eventuali spurious wake up ed altri problemi, garantendoci che quando si sveglia, se esce dal while, è perché ha un risultato concreto.

Vediamo un ultima versione utilizzando una BlockingQueue:

```
static int threadRace(Runnable a, Runnable b) {
  BlockingQueue<Integer> result = new ArrayBlockingQueue<>(2);

  class MyThread extends Thread throws InterruptedException {
     int id;
     Runnable r;
     MyThread(Runnable r, int id) { /*solito costruttore*/ }

     public void run() {
        // Richiamo il run del runnable associato a questa classe
        r.run();

        // put e' bloccante e siccome run non puo' lanciare IE devo gestirla
        try {
           result.put(id);
        } catch(InterruptedException e) {
           return;
        }
```

```
19      }
20    }
21
22    Thread t1 = new MyThread(a, 1), t2 = new MyThread(b, 2);
23
24    t1.start();
25    t2.start();
26
27    // La blockingQueue mi assicura di ritornare solo quando avro' un elemento
28    return result.take();
29  }
```

20.2.2 Alcune note sulla concorrenza

Il docente, per chi volesse approfondire argomenti di concorrenza che non vengono trattati nel corso, consiglia il libro *Java Concurrency in practice* evidenziandolo come *"la bibbia della concorrenza in Java"*, il libro è vecchio di 10 anni e manca di coprire alcuni argomenti di livello più alto, ma è ancora ottimo punto di riferimento, parlando di questi argomenti facciamo un elenco a livelli:

1. Parallel Streams (livello più alto), aggiunti in Java 8, lavorano con le lambda espressioni e facilitano creando una parallelizzazione automatica
2. Fork-join e ExecutorService, aggiunti in Java 7 sono framework che permettono di parallelizzare calcoli, come ad esempio trovare il massimo in un array di un miliardo di elementi, la fork parallelizza e la join fa il merge del risultasto tra i vari thread, una sorta di "dividi-e-fondi", mentre ExecutorService maschera la creazione di Thread prendendo in ingresso un Runnable
3. BlockingQueue e Wrappers Atomici
4. Synchronize, wait/notify e Thread (livello più basso)

20.3 Java Memory Model

È un modello di memoria, ovvero l'insieme delle regole che descrivono come un architettura gestile la memoria, siccome la JVM è un architettura virtuale ha il suo memory model che converte ed emula tramite l'architettura reale sottostante.

Una buona conoscenza del JMM permette di scrivere codice portabile, ed è composto da tre regole principali:

- Atomicità
- Visibilità
- Ordinamento

20.3.1 Regole di atomicità (JMM)

Le regole di atomicità dicono quali istruzioni sono atomiche in Java in assenza di sincronizzazione esplicita, ovvero che non possono essere interrotti da altri thread, le regole sono le seguenti:

- Una singola lettura e scrittura di una variabile di tutti i tipi, tranne long e double
- Una singola lettura e scrittura di una variabile dichiarata `volatile`

La prima regola è dovuta al fatto che la JVM deve garantire queste regole, però siccome i long e double sono tipi di dato a 64 bit in alcuni sistemi a 32 bit è necessario accedere a 2 registri, questo rende l'istruzione non atomica.

La keyword `volatile` si può applicare unicamente ad attributi di una classe, ha l'effetto di rendere atomica una lettura e scrittura anche per long e double, un campo volatile non può essere final, questa keyword ha anche altre caratteristiche che saranno approfondite successivamente.

Vediamo alcuni esempi per chiarire queste regole di atomicità:

```
1  int a, b;
2  double d;
3  a = 5; // Atomica
4  a = b; // Non atomica, lettura + scrittura
5  d = 42; // Non atomica, d e' un double
6
7  Object x, y;
8  x = null; // Atomica
9  x = y; // Non atomica, lettura + scrittura
10
11 volatile long c, e;
12 c = 74; // Atomica, c e' volatile
13 c = e; // Non atomica, lettura + scrittura
```

21 Lezione 21

La lezione 21 continua con la JMM con le regole di ordinamento e visibilità, la Lazy Initialization e due esercizi.

21.1 Regole di visibilità (JMM)

Si veda il seguente esempio:

```
1  public class ThreadVisibility {
2    private static boolean done;
3    private static int n;
4
5    public static void main(String args[]) {
6       Thread t = new Thread() {
7          public void run() {
8             n = 42;
9             try {
10               sleep(1000);
11            } catch(InterruptedException e) {
12               return;
13            }
14            System.out.println("Fatto");
15            done = true;
16         }
17      }
18   }
19
20   t.start();
21   while(!done) { }
22   System.out.println(n);
23 }
```

Possiamo vedere `done` inizializzato a falso a riga 2, un attesa attiva a riga 21, ed un thread, l'output che qualcuno potrebbe aspettarsi è "Fatto" e "42", eppure quando si compila ed esegue il programma stampa "Fatto" e rimane in loop.

Questo capita in quanto in mancanza di sincronizzazione **non ci sono garanzie di visibilità** che quello che fa un thread sia visibile da altri thread, di default ogni thread *"si fa i fatti suoi"* e le modifiche non diventano visibili su altri thread anche per variabili condivise.

In questo caso non ci sono garanzie perché il thread che scrive su `done` non lo legge più, di conseguenza il compilatore e la JVM per ottimizzazione non eseguono proprio quell'istruzione, perché per quel thread è inutile farlo, infatti questo potrebbe accadere anche per l'istruzione `n = 42`, ma si noti che anche mettendo un ipotetica println per stampare i valori, questo continua a **non dare alcuna garanzia**, per avere garanzie è necessario utilizzare la sincronizzazione.

I principi fondamentali della visibilità inter-thread sono:

- In mancanza di sincronizzazione, le operazioni di scrittura svolte da un thread possono rimanere nascoste ad altri thread **a tempo indefinito**

- In particolare, alcune operazioni possono rimanere nascoste ed altre visibili

La visibilità è garantita solamente dalle seguenti operazioni:

- Acquisire un monitor rende visibili le operazioni effettuate dall'ultimo thread che possedeva quel monitor fino al suo rilascio
- Leggere il valore di una variabile volatile rende visibili le operazioni effettuate dall'ultimo thread che ha modificato quella variabile (anche se non riguardano strettamente quella variabile)
- Invocare `t.start()` rende visibili al nuovo thread tutte le operazioni effettuate dal chiamante
- Ritornare da una invocazione di `t.join()` rende visibili tutte le operazioni effettuate dal thread t fino alla sua terminazione

Si noti che le classi thread safe come BlockingQueue utilizzano queste regole.

Una soluzione al problema iniziale è dichiarare `done` come volatile, possiamo dire che il corollario di volatile è *"Poter leggere sempre l'ultimo valore di quella variabile assegnato da qualsiasi thread"*, si noti che al momento della modifica di una variabile volatile si verranno a sapere **tutte** le scritture precedenti, quindi non c'è necessità di creare 10 variabili volatile ma modificare normalmente le variabili e poi aggiornarne una volatile, così facendo l'altro thread potrà accedere subito ai valori aggiornati, si noti che nello stesso esempio si può vedere come n venga assegnato 42, non sia volatile, ma abbia il valore aggiornato.

Si noti che volatile è una keyword costosa in performance di tempo, in quanto richiede un invalidazione di registri ed aggiornamento di essi, quindi è buona norma non utilizzarla in modo inappropriato.

Sia synchronized che volatile offrono **garanzie di atomicità e visibilità**, volatile come visto inoltre rende atomiche piccole operazioni (singole letture o scritture su long e double), ma è un vantaggio minore e viene più utilizzata per la visibilità, per vantaggi di atomicità il 99% delle volte si vuole utilizzare un blocco synchronized, ovviamente non parliamo di operazioni che diventano letteralmente atomiche, ma che la mutua esclusione offre un vantaggio simile ad un operazione atomica.

21.2 Regole di ordinamento (JMM)

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;`

Da quanto abbiamo visto possiamo dedurre vari output per le variabili r1 ed r2, ovvero:

- r1 = 0, r2 = 1, se Thread 1 viene eseguito per primo
- r1 = 1, r2 = 0, se Thread 2 viene eseguito per primo
- r1 = 0, r2 = 0, se lo scheduler interrompe un thread durante le assegnazioni

- r1 = 1, r2 = 1, sebbene sembri un risultato assurdo, la JMM consente anche questo risultato

Difatti, in mancanza di sincronizzazione, al compilatore ed alla JVM è consentito **riordinare le istruzioni**, a patto che tale riordino sia ininfluente dal punto di vista del singolo thread, infatti proprio per questo è consentito invertire l'ordine delle istruzioni di Thread 1 o Thread 2, se l'ordine viene invertito il risultato r1 = 1, r2 = 1 diventa possibile.

Questa libertà di eseguire le istruzioni in ordine diverso da come sono state scritto è causato da parte di ottimizzazioni del processore che prende istruzioni tra di loro non connesse e le esegue in blocchi simili da aumentare le performance.

I costrutti synchronized e volatile riducono le possibilità di riordino, prese due istruzioni successive x ed y, e supponiamo non abbiano dipendenze nel thread singolo, distinguiamo le seguenti 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

Fatta questa distinzione vediamo la seguente tabella:

Tipo x/y	Istruzione normale	Lettura volatile Inizio synchronized	Scrittura volatile Fine synchronized
Istruzione normale	**Sì**	**Sì**	No
Lettura volatile Inizio synchronized	No	No	No
Scrittura volatile Fine synchronized	**Sì**	No	No

In pratica abbiamo che:

1. Le istruzioni normali si possono sempre scambiare
2. Le istruzioni normali possono entrare dentro un blocco sincronizzato
3. Le istruzioni normali che precedono la lettura di una volatile possono spostarsi dopo la lettura
4. Le istruzioni normali che seguono la scrittura di una voltatile si possono spostare prima della scrittura

Insomma la JVM può allargare un blocco sincronizzato ma nulla può uscire da tali blocchi, ovviamente nel blocco le istruzioni tra di loro possono a loro volta scambiarsi.

21.2.1 Esercizio SimpleThread (Regole ordinamento)

Dato il seguente pezzo di codice ed ipotizzando che partano contemporaneamente due istanze di SimpleThread, dire tutti i possibili output:

```
1 public class SimpleThread extends Thread {
2   private static volatile int n = 0;
3   public void run() {
4      n++;
```

```
    int m = n;
    System.out.println(m);
  }
}
```

I possibili output sono:

1. Il caso più facile è che viene eseguito prima completamente uno e poi l'altro, in quel caso avremo 1 e 2 come output.
2. Si può ottenere anche 1 e 1 in quanto n++ non è un istruzione atomica, e mentre uno la legge l'altro la modifica.
3. Si può ottenere anche 2 e 1, per il primo motivo.
4. Anche 2 e 2 è un output possibile se il secondo parte dopo che n++ è stato eseguito, finisce e poi torna indietro
5. Non è possibile ottenere un thread che stampi 0, tutte le operazioni sono dipendenti tra di loro

21.3 Lazy Initialization

Possiamo quasi definirlo un pattern, ed è un problema ricorrente nella programmazione ad oggetti, ipotizziamo di voler effettuare l'istanzazione un singleton, ovvero un oggetto singolo, ma che questo oggetto venga istanziato solo se richiesto, per questo prende il nome di *"inizializzazione pigra"*.

Un approccio ingenuo potrebbe essere:

```
class A {
  private static HeavyClass special;

  public static HeavyClass getSpecial() {
     if (special == null) {
        special = new HeavyClass();
     }
     return special;
  }
}
```

Ma questa implementazione non è thread safe e presenta due problemi:

1. Problema 1: Due thread invocano contemporaneamente getSpecial, possono essere creati **due** oggetti HeavyClass, questo è dovuto a mancata atomicità della sequenza lettura di special, scrittura di special.
2. Problema 2: Due thread invocano contemporaneamente getSpecial, 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, questo è dovuto alle regole di visibilità del JMM.

La soluzione più semplice, ma non efficiente, è dichiarare synchronized il metodo getSpecial e risolvere entrambi i problemi, è poco efficiente perché mette un overhead di performance (prendere un monitor è costoso) su **tutte** le invocazioni di getSpecial, anche molto tempo dopo l'inizializzazione dell'oggetto quando non è più necessaria la sincronizzazione.

La soluzione più avanzata ed efficiente è quella di utilizzare una classe interna che "inscatola" (o fa wrapping) il riferimento a quell'elemento, questo sfrutta la caratteristica di Java che le classi interne sono caricate in modo dinamico quando vengono utilizzate in runtime, quindi il new avviene solo in quel caso, questo è inoltre thread safe in quanto la JVM garantisce che l'inizializzazione di classi è atomica.

```
class A {
  private static class HeavyClassHolder {
     static HeavyClass special = new HeavyClass();
  }

  public static HeavyClass getSpecial() {
     returnHeavyClassHolder.special;
  }
}
```

21.4 Esercizio Missing synchronized (synchronize)

Questa soluzione riguarda la traccia "missing synch 3" del 2019-03-19.

(a) No, perché possono sovrapporsi le operazioni di scambio e possono accadere cose strane.

(b) No, perché ogni thread avrà un this diverso, e quindi monitor diverso, e non offre benefici

(c) Equivalente a (b)

(d) Errore di compilazione, non è consentito utilizzarlo così senza oggetti

(e) Compila, diventa corretto e sequenzializza completamente l'operazione, solo che è poco efficiente

(f) Corretto, è un oggetto condiviso tra i due oggetti

(g) Assolutamente errato, *"quasi una battuta"* in quanto non grstisce nemmeno l'InterruptedException ed altre cose bizzarre

(h) Inutile

(i) Errore di compilazione, perché A[i] è un primitivo, anche se fosse un riferimento *"Mi esporrei ad incubi notturni perché sto modificando il monitor stesso"*

(j) Corretto e funzionante, più efficiente di (e) ed (f) in quanto i thread lavorano insieme rispetto ai precedenti che lo rendevano sequenziale (annullando in parte il vantaggio del multithreading)

(k) Anche questo corretto

21.5 Esercizio VoteBox (thread in generale)

"I server del dieti bruciati per un blackout di 5 secondi, e noi stiamo qui a parlare del Java Memory model, ah, e state anche registrando il tutto, adesso non fatemi licenziare per insubordinazione"

- M. Faella

```
class VoteBox {
  private final int maxVoti;
  private int votoTrue, votoFalse;
  private Set<Thread> votanti = new HashSet<>();

  public VoteBox(int numT) {
     if(numT <= 0) {
        throw new IllegalArgumentException(...);
     }
     maxVoti = numT;
  }

  public void vote(boolean v) {
     synchronized(votanti) {
        if(!votanti.add(Thread.currentThread()) {
           throw new IllegalStateException(...);
        }

        if(votantiSize() == maxVoti) {
           return;
        }

        if(v) {
           votiTrue++;
        } else {
           votiFalse++;
        }
        votanti.notifyAll();
     }
  }

  // Questo metodo e' pubblicamente bloccante, quindi per la disciplina delle
     interruzioni lo dichiariamo come tale
  public boolean waitForResult() throws InterruptedException {
     synchronized(votanti) {
        while(votanti.size() != maxVoti) {
           votanti.wait();
        }
        return (votiTrue > votiFalse);
     }
  }

  public boolean isDone() {
     // Essenziale per la visibilita' tra i thread, o questo oppure volatile
     synchronized(votanti) {
        return votanti.size() == maxVoti);
     }
  }
}
```

22 Lezione 22

La lezione 22 parla un po' della Java GUI ed una manciata di patterns.

22.1 Java GUI

Le GUI sono un altro di quegli aspetti in cui Java fu pioniere nella sua creazione, furono una delle funzionalità della libreria standard grazie alla JVM.

Esempio di creazione di una semplice finestra vuota in Java Swing:

```
1  import javax.swing.*;
2
3  public class EmptyFrame {
4    public static void main(String[] args) {
5       final int WIDTH = 200, HEIGHT = 200;
6       // Creo la finestra
7       JFrame frame = new JFrame();
8       // Imposto le dimensioni iniziali
9       frame.setSize(WIDTH, HEIGHT);
10      // Imposto l'operazione di chiusura
11      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12      // Mostro la finestra
13      frame.setVisible(true);
14   }
15 }
```

Il risultato è una finestra vuota, si noti che il programma non è terminato anche se il main lo è, questo perché viene creato un nuovo thread in modo implicito che si occupa degli eventi della GUI, questo thread termina soltanto premendo "X" e chiudendo la finestra.

Vediamo un esempio più concreto, ovvero 2 bottoni cliccabili ma che non fanno niente ed una textfield editabile:

```
1  import javax.swing.*;
2
3  public class FrameTest {
4    public static void main(String[] args) {
5       // Creo la finestra
6       JFrame frame = new JFrame();
7
8       // Creo due pulsanti
9       JButton helloButton = new JButton("Say Hello");
10      JButton goodbyeButton = new JButton("Say Goodbye");
11
12      // Creo un campo di testo
13      final int FIELD_WIDTH = 20;
14      JTextField textField = new JTextField(FIELD_WIDTH);
15      textField.setText("Click a button!");
16
```

```
     // Ottengo un riferimento al "pannello del contenuto (contentPane) della
        finestra
     Container contentPane = frame.getContentPane();

    // Imposto il layout del pannello
    contentPane.setLayout(new GridLayout(2, 2));

    // Aggiungo al pannello i 3 elementi che ho creato
    contentPane.add(helloButton);
    contentPane.add(goodbyeButton);
    contentPane.add(textField);

    // Imposto l'operazione di chiusura
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Stabilisco le dimensioni iniziali in base al layout
    frame.pack();
    // Mostro la finestra
    frame.setVisible(true);
  }
}
```

Allo stato attuale, la pressione dei pulsanti non provoca nessuna reazione nell'applicazione, attiviamo ora questi widget in modo che accada qualcosa, per farlo dobbiamo implementare `ActionListener` che è un interfaccia con un unico metodo `actionPerformed`, si noti che questa è un interfaccia funzionale e quindi offre tutti i vantaggi già visti.

La sua dichiarazione è la seguente:

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

Il primo metodo per fare questo, in maniera più esplicita, è implementare una classe locale che implementa l'interfaccia, modifichiamo il main come segue:

```
 // Creo la finestra, etc.
 ...

// Creo un campo di testo
final int FIELD_WIDTH = 20;
JTextField textField = new JTextField(FIELD_WIDTH);
textField.setText("Click a button!");

// Classe locale che gestira' gli eventi
class Ascoltatore implements ActionListener {
  private String msg;

  public Ascoltatore(String s) {
     msg = s;
  }

```

```
17    public void actionPerformed(ActionEvent event) {
18    textField.setText(msg);
19    }
20  }
21
22  // Creo il primo pulsante
23  JButton helloButton = new JButton("Say Hello");
24  // Associo un oggetto osservatore al pulsante
25  helloButton.addActionListener(new Ascoltatore("Hello, World"));
26
27  // Faccio lo stesso per il secondo pulsante
28  JButton goodbyeButton = new JButton("Say Goodbye");
29  goodbyeButton.addActionListener(new Ascoltatore("Goodbye, World!"));
```

22.2 Pattern Observer

Ogni pattern è identificato da un nome che identifica lo schema di progettazione, il contesto ed una soluzione, questi pattern sono generici per poter essere adattati a qualsiasi linguaggio di programmazione.

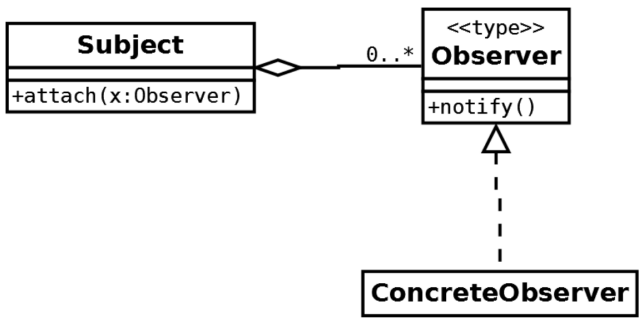
Contesto del pattern:

1. Un oggetto (soggetto) genera eventi
2. Uno o più oggetti (osservatori) vogliono essere informati del verificarsi di tali eventi

Soluzione del pattern:

1. Definire un interfaccia "Observer" con un metodo convenzionalmente chiamato "notify" che sarà implementato dagli osservatori
2. Il soggetto ha un metodo che gli consente di registrare osservatori, convenzionalmente chiamato "attach"
3. Il soggetto gestisce l'elenco dei suoi osservatori registrati
4. Quando il soggetto genera un evento quest'ultimo informa tutti i suoi osservatori registrati invocando il loro metodo notify

Ad ogni pattern è solitamente associato un diagramma UML, in questo caso è il seguente:



In questo diagramma è presente la classe Subject (soggetto osservato) e l'interfaccia Observer (osservatore), si noti la relazione di **aggregazione uno a molti** tra Subject ed Observer, tale relazione indica che ogni oggetto di tipo Subject conserva un insieme di riferimenti ad oggetti di tipo Observer, ovvero tiene traccia degli osservatori che si sono registrati chiamando attach.

Riconosciamo questo pattern nell'esempio di GUI fatto precedentemente, il soggetto era il JButton mentre ActionListener l'interfaccia concreteListener della classe, si noti che ActionListener al suo interno prende informazioni sull'evento.

La libreria standard offre un supporto per implementare questo pattern, ovvero avviene tramite l'interfaccia `Observer` e la classe `Observable` del package java.util, l'interfaccia Observer offre il metodo `update(Observable o, Object arg)`, il primo parametro è il soggetto che genera eventi mentre il secondo è l'osservatore, questo metodo corrisponde a notify.

```
1 public interface Observer {
2   void update(Observable o, Object arg);
3 }
```

Mentre la classe `Observable` offre il metodo `addObserver` che si occupa di registrare un osservatore e corrisponde al metodo attach, ed il metodo `notifyObservers` che invoca l'update di tutti gli osservatori registrati, passando this come primo argomento e arg come secondo.

```
1 public class Observable {
2   public void addObserver(Observer o) { ... }
3   public void notifyObservers(Object arg) { ... }
4 }
```

L'uso tipico di questa classe consiste nell'estenderla, in modo da aggiungere ad una nostra classe la capacità di essere osservata, altre classi implementeranno Observer e fungeranno da osservatori, questo ci libera dal dover implementare il meccanismo di registrazione e notifica degli osservatori.

22.3 Paradigma Model-View-Controller

Il paradigma MVC consiste nel distinguere, all'interno di un sistema informatico, tre tipologie di componenti:

- Quelle che presentano i dati in oggetto (Model)
- Quelle che si occupano di presentare i dati all'utente (View)
- e quelle che si occupano dell'interazione con l'utente (Controller)

Il modello rappresenta i dati in sé, indipendentemente da come essi vengono visualizzati, il modello non deve dipendere né dalle viste né dai controller con cui interagisce, inoltre deve offrire le seguenti funzionalità:

- Per le viste, deve offrire metodi che espongono i dati in sola **lettura**
- Per i controller, deve offrire metodi che permettano di modificare i dati in risposta a delle richieste dell'utente
- Infine, in caso di cambiamento dei dati, deve avvisare le viste, affinché queste possano rispecchiare il cambiamento

- Quest'ultima funzionalità viene tipicamente realizzata tramite il pattern Observer

Quindi il controller non dialoga direttamente con le viste, ma abbiamo il controller che comunica con il modello e quest'ultimo aggiorna le viste.

22.4 Pattern Strategy

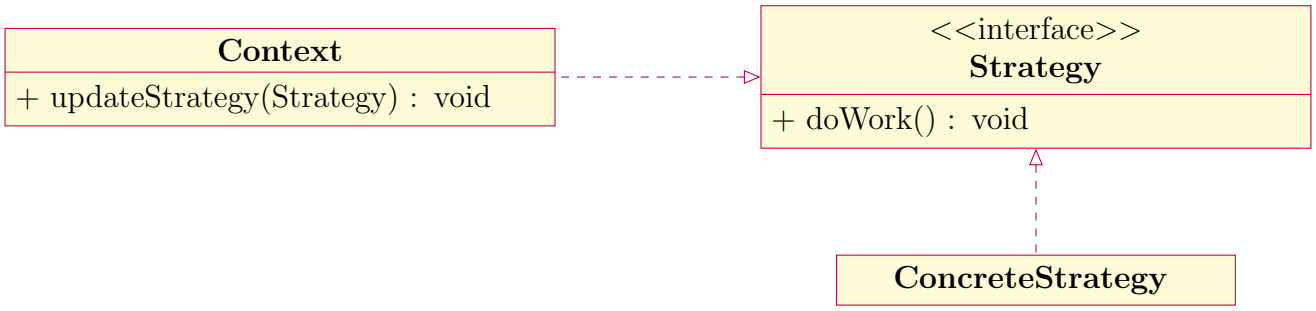
Uno dei pattern classici, è già stato usato nell'esempio sulle GUI, si occupa delle situazioni in cui una classe accetta una procedura come argomento. **Contesto del pattern:**

1. Una classe (context) può sfruttare diverse varianti di un algoritmo
2. I clienti della classe vogliono fornire versioni particolari dell'algoritmo

Soluzione del pattern:

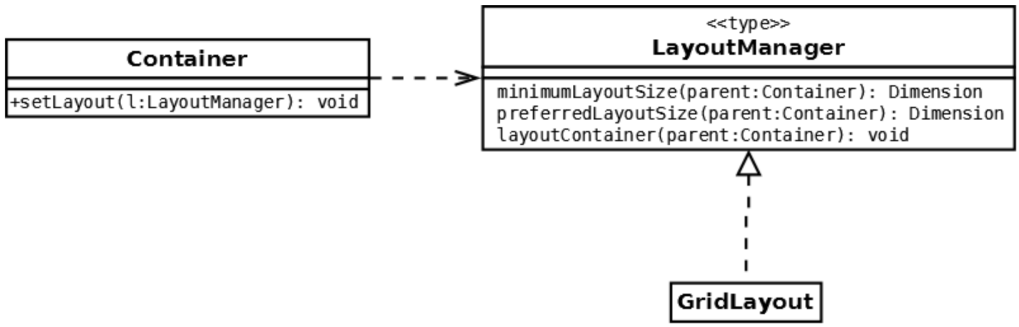
1. Definire un interfaccia (Strategy) che rappresenti un'astrazione dell'algoritmo
2. Per fornire una variante, il cliente costruisce un oggetto classe concreteStrategy che la implementa l'interfaccia Strategy e la passa alla classe context
3. Ogni volta che deve eseguire l'algoritmo, la classe context invoca il corrispondente metodo dell'oggetto che concretizza la strategia

Il suo diagramma UML è il seguente:



Questo pattern si può vedere con il metodo `.setLayout()` che può accettare GridLayout oppure FlowLayout.

Il punto forte di questo pattern è la scalabilità, ovvero posso crearmi un layout mio e poi darlo in ingresso e funzionerà comunque, rispetto al C dove setLayout erano massimo 3 e prendeva in ingresso un intero per stabilire quale utilizzare, il diagramma riguardo il LayoutManager:



22.5 Pattern Composite

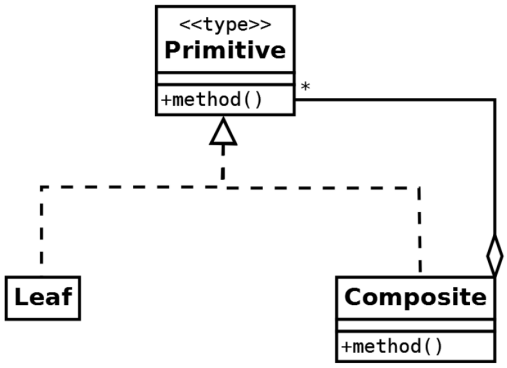
È un pattern che consente di creare alberi di oggetti, dove gli oggetti più complessi sono nodi mentre quelli primitivi sono foglie. **Contesto del pattern:**

1. Oggetti primitivi possono essere combinati in un oggetto composito
2. I client possono trattare un oggetto composito come primitivo, ovvero questo oggetto può essere aggiunto ad altri oggetti compositi

Soluzione del pattern:

1. Definire un interfaccia (Primitive) che rappresenti un'astrazione dell'oggetto primitivo
2. Un oggetto composito contiene una collezione di oggetti primitivi
3. Sia gli oggetti primitivi che quelli compositi implementano l'interfaccia Primitive
4. Nel realizzare un metodo dell'interfaccia Primitive, un oggetto composito applica il metodo corrispondente a tutti i propri oggetti primitivi e poi combina i risultati ottenuti

Il suo diagramma UML è il seguente:



Primitive è l'interfaccia che rappresenta un oggetto primitivo, sia i veri oggetti primitivi (chiamati leaf) sia gli oggetti compositi implementano l'interfaccia primitive, la relazione di aggregazione tra composite e primitive indica che un oggetto di tipo composite contiene un insieme di riferimenti ad oggetti primitivi.

22.6 Pattern Decorator

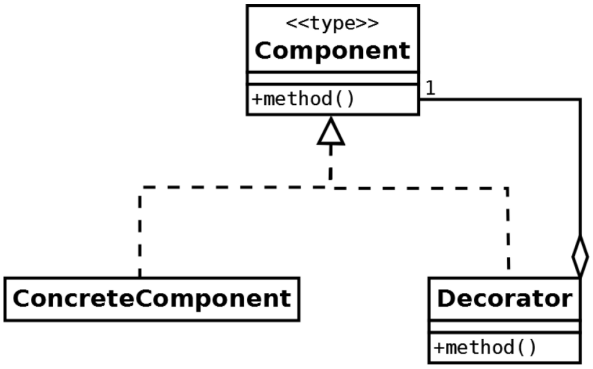
È un pattern che consente di aggiungere funzionalità senza coinvolgere la classe, un esempio può essere voler aggiungere una barra di scorrimento ad una JTextArea. **Contesto del pattern:**

1. Si vuole decorare (ovvero *migliorare*, ovvero *aggiungere funzionalità*) ad una classe componente
2. Un componente decorato può essere utilizzato allo stesso modo di uno normale
3. La classe componente non vuole assumersi la responsabilità della decorazione
4. L'insieme dele decorazioni possibili non è limitato

Soluzione del pattern:

1. Definire un'interfaccia (Component) che rappresenti un'astrazione di un componente
2. Le classi concrete che definiscono componenti implementano l'interfaccia Component
3. Definire una classe (Decorator) che rappresenta la decorazione
4. Un oggetto decoratore contiene e gestisce l'oggetto che decora
5. Un oggetto decoratore implementa l'interfaccia Component
6. Nel realizzare un metodo di Component, un oggetto decoratore applica il metodo corrispondente all'oggetto decorato e ne combina il risultato con l'effetto della decorazione

Il suo diagramma UML è il seguente:



L'interfaccia Component rappresenta un componente generico, sia i veri componenti base che le loro decorazioni implementano l'interfaccia, in modo che il client li possa usare nello stesso modo. La relazione di aggregazione indica hce un oggetto decoratore contiene un riferimento al componente che sta decorando.

22.7 Differenze tra Composite e Decorator

Confrontando le descrizioni dei diagrammi dei pattern Composite e Decorator notiamo diverse somiglianze, entrambi prevedono una distinzione tra oggetti di base (primitivi e componenti) e oggetti composti ed entrambi prevedono che due categorie implementino un interfaccia comune.

Tuttavia, i contesti di applicazione sono diversi:

- **Composite**: Si applica quando bisogna **aggregare più oggetti** di base in un unico oggetto che può essere a sua volta aggregato con altri, ottenendo una gerarchia di oggetti disposti in un albero
- **Decorator**: Si applica quando bisogna **aggiungere funzionalità**, illimitate a priori, ad una classe senza coinvolgerla in maniera diretta

Lista dei teoremi, definizioni ed osservazioni