A.A. 2021-2022

Linguaggi di Programmazione II

Università degli studi di Napoli

Federico II

Valentino Bocchetti - N86003405

Indice

1 Introduzione al corso

Lo scopo del corso è quello dello studio dei linguaggi di programmazione in maniera applicata (toccando anche campi propri di INGSW).

Si utilizzeranno principalmente **Java** e **C++**.

1.1 Testi consigliati

- Core Java - Horstmann (sufficiente il primo volume);
- Java Precisely - Sestoft;
- Seriously Good Software - (del prof, offerto dal prof).

1.2 Contatti

Per contattare il prof inviare una mail qui

Ricevimento → Lunedì dalle 14:00 alle 16:00.

1.3 Esercizi

Di seguito un link all'archivio degli esercizi.

1.4 Esame

L'esame consiste in una prova intercorso (che vale per i primi `2/3` appelli e che influisce per ca il `50%` dello scritto) e una prova scritta.

Con la prova orale:

- Ammessi;
- Non ammessi;
- Esonerati.

2 Lezione del 07-03

2.1 Tipi primitivi

I 4 tipi interi hanno questa catena di promozione:

$$long \leftarrow int \leftarrow short \leftarrow byte$$

I 2 tipi a virgola mobile invece:

$$double \leftarrow float$$

Abbiamo ancora:

$$int \leftarrow char$$

$$float \leftarrow long$$

$$double \leftarrow long$$

Questo tipo di conversione presenta perdita di informazioni (un esempio è da **long** a **float** o da **int** a **float**).

Il caso da **int** a **float** è dovuto al problema della mantissa dei float. Infatti entrambi i tipi utilizzano 32 bit, ma negli int questi vengono utilizzati tutti per il valore (per il float invece no).

Il tipo `boolean` invece è isolato.

2.1.1 Esercizio 1

Scrivere e inizializzare a in modo tale che sia un ciclo infinito

```
float a = 1E9; // un miliardo
while(a == a + 1){
    // Ciclo infinito
}
```

Se utilizzassi un intero assegnando il suo valore alla classe wrapper `Integer.MAX_VALUE`, abbiamo un wrapper round (intero più piccolo).

2.1.2 Esercizio 2

Scrivere quante volte viene eseguito il ciclo:

```
for(double x = 0; x != 1.0; x += 0.1){
    // Viene eseguito infinite volte
    // 0.1 non è possibile rappresentarlo in un double (è un numero periodico)
    // Per ovviare a questo problema si cambia la condizione da != a <=
}
```

2.2 Diagrammi di memory layout

Rappresentazione grafica dello stato della memoria in corrispondenza di un cambio di codice.

es.

```
// Classe che contiene 2 campi: `String name` e `Int salary`
Employee e = new Employee("Pippo", 2500);
```

Il salario essendo un tipo primitivo verrà contenuto in una zona di memoria, mentre la stringa Pippo verrà tenuto in una zona di memoria accessibile dal puntatore che contiene corrispondente al nome.

L'oggetto *e* verrà allocato sullo **`stack`**, mentre la scatola verrà allocata nello **`heap`** (ci sarebbe da considerare anche il caso particolare della stringa, che però per brevità ignoreremo).

2.2.1 Memory layout di un frammento di codice - Esercizio

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

3 Lezione del 11-03

3.1 Eccezioni `checked/unchecked`

Possiamo confrontare queste 2 categorie di eccezioni su vari piani:

1. Differenza a compile time
2. Differenza per la JVM;
3. Differenza in base alla gerarchia;
4. Pragmatica della distinzione → Quale scelta effettuare per sollevare un errore.

Sulla differenza a compile time, il compilatore obbliga il programmatore a gestire l'eccezione (checked):

- Mediante un blocco **`try/catch`**;
- Mediante la keyword **`throws`**
  - Che può essere anche sostituito con la chiamata a un metodo che contenga nella firma un **`throws`** di una eccezione checked;
  - Mettendo nella firma del metodo un **`throws`** (dell'eccezione che la funzione può lanciare o di una sua superclasse).

Origine

	throw	metodo `g()`
try … catch	Questo è un errore logico (Se il metodo lo sa gestire non dovrebbe lanciare l'eccezione)	So trattare l'anomalia
throws	OK	Non so trattare l'eccezione (posso propagare l'eccezione fino al main, che a sua volta può rilanciare l'eccezione)

Sulla differenza per la JVM, non esiste nessuna differenza. La distinzione vive solo a tempo di compilazione.

Sulla differenza in base alla gerarchia vale un discorso mnemonico. La radice delle eccezioni è `Throwable`, che di default è `checked` (un oggetto si può lanciare solo se estende `Throwable`). Proprio per questo motivo tutte le eccezioni sono checked.

A sua volta `Throwable` si divide in:

- `Error` → Di default è `unchecked`
- `Exception`
  - Un figlio fondamentale delle Exception è `RunTimeException` (che è di tipo `unchecked`).

Seguendo il discorso dell'estensione, se estendo una classe checked la mia eccezione custom sarà checked (stesso discorso per le eccezioni unchecked).

3.1.1 Linee guida per la scelte delle eccezioni

Per buona norma tutte le eccezioni `unchecked` sono quelle evitabili → Cattiva implementazione da parte del programmatore. Tutti gli altri casi invece dovrebbero essere `checked` (Ci si trova quindi in errori dovuti a fattori esterni).

Esempi canonici per la gestione delle eccezioni checked:

```
1 FileNotFoundException f;
2
3 RuntimeException r; // ArrayIndexOutofBoundary
```

3.2 Il sistema dei Tipi

I tipi previsti da Java sono:

- Tipi primitivi (8);
- Tipi di categoria riferimento (classi che rappresentano interfacce)
  - Array

- Tipo nullo → unico valore `null`.

Sulla base di questo sistema di tipi si definisco delle relazioni di compatibilità. Tra quelle viste in precedenza ricordiamo la relazione di compatibilità (promozione).

3.2.1 Relazione di sottotipo

Relazione di compatibilità di tipi non primitivi.

∀ tipo `T`, `U` non primitivi:

1. `T` è sottotipo di se stesso (riflessiva);
2. `T` è sottotipo di `Object`;
3. Se `T` estende o implementa (anche indirettamente) `U`, `T` è sottotipo di `U` (antisimmetrica e transitiva);
4. Il tipo `nullo` è sottotipo di `T`;
5. Se `T` è sottotipo di `U`, allora `T[]` è sottotipo `U[]` (gli array preservano la relazione di sottotipo)

Le prime 4 regole sono di tipo `ground`. La 5 è di tipo `ricorsiva`.

3.2.2 Relazione di assegnabilità

Dati 2 tipi qualsiasi `T` e `U`, `T` è assegnabile a `U` se e soltanto se:

1. `T` e `U` sono primitivi e esiste conversione implicita `T` → `U`;
2. `T` e `U` sono entrambi non primitivi e `T` è sottotipo di `U`.

NB: Queste regole non tengono conto dell'auto-unboxing, dell'autoboxing e del polimorfismo parametrico (i Generics in Java e i Template in `C++`).

es.

1. Assegnamento → `id = exp`
    - *Employee e = new Manager()*
        - Type checking in Java → `exp` è assegnabile a `id` solo se il tipo di `id` è assegnabile al tipo di `exp`
2. `return expr`
    - Compila correttamente quando il tipo di expr è assegnabile al tipo di ritorno del metodo.
3. Invocazione di metodo
    - Compila correttamente quando i parametri attuali sono assegnabili ai parametri formali del metodo.

3.2.3 Operatore `instanceof`

Operatore binario infisso:

`exp instanceof T`

È valido se il tipo effettivo di `exp` è sottotipo di `T` → L'eccezione è il tipo `null` (`instanceof` in questo caso restituisce null).

La definizione completa quindi sarà:

Il tipo effettivo di `exp` è sottotipo di `T` e non è nullo

NB: Il tipo effettivo non può essere un'interfaccia (che è un tipo dichiarato).

3.3 Esercizi

```
1  String[] a = new String[10]; // Inizializza un'array di 10 stringhe inizializzati a null
2  Object[] x = a;
3
4  /*
5     Il tipo dichiarato è Object
6     Il tipo effettivo di x[0] è null prima dell'assegnazione.
7     Il codice compila
8
9     A RunTime viene lanciata un'eccezione -> Tutte le scritture in array (in Java mantengono un
          informazione della loro lunghezza e del tipo con il quale sono stati dichiarati)
10    a RunTime vengono controllate dal type checker (In questo caso violo le dichiarazioni precedenti)
11 */
12
13 x[0] = new Object();
14 String s = a[0]; // a[0] punta ad un Object
```

3.4 Conversione esplicite di tipo: Cast

Java permette alcune conversioni esplicite tramite *cast*.

Si può utilizzare un cast per effettuare esplicitamente una promozione (il cast risulta in questo caso superfluo).

Si può utilizzare un cast per effettuare una **promozione al contrario**. È sconsigliato utilizzare un downcast (tipi primitivi); è preferibile infatti fare uso dell'apposita classe **Math**.

Sono consentiti dal compilatore i seguenti cast tra un tipo riferimento (o array) `A` ad un tipo riferimento (o array) `B`:

- Se `B` è supertipo di `A` → Si chiama `upcast`, ma è superfluo;

- Se `B` è sottotipo di `A`
  - Si chiama `downcast`;
  - A run-time, la JVM controlla che l'oggetto da convertire appartenga effettivamente ad una sottolasse di `B` (in caso contrario, viene sollevata l'eccezione ClassCastException);
  - Si deve cercare di evitare i downcast, perché aggirano il type checking svolto dal compilatore (possono essere utilizzati a questo scopo i tipi parametrici);
  - Se si è costretti a usare un downcast, va preceduto da un controllo `instanceof`, che assicuri la correttezza della conversione.

Negli altri casi invece il cast porta ad un errore di compilazione.

3.5 Tipi wrapper

Per ogni tipo base, Java offre una corrispondente classe, che ingloba un valore di quel tipo in un oggetto. Sono **immutabili** e **final**.

Ogni classe wrapper ha un **costruttore** che accetta un valore del tipo base corrispondente.

Ogni classe wrapper ha un metodo statico valueOf che prende come argomento un valore del tipo base corrispondente alla classe e restituisce un oggetto wrapper che lo ingloba:

- A differenza del costruttore, l'oggetto restituito non è necessariamente nuovo;
- ovvero, il metodo valueOf cerca di riciclare gli oggetti già creati (caching);
- in generale, questo non è un problema, perché gli oggetti wrapper sono immutabili.

Le sei classi wrapper relative ai tipi numerici estendono la **classe astratta Number**.

3.5.1 Autoboxing e auto-unboxing

L'autoboxing può convertire un'espressione di tipo primitivo in un oggetto del tipo wrapper corrispondente, o di un suo supertipo.

3.5.2 Tipi wrapper e uguaglianza

L'operatore == può dare risultati inaspettati se applicato ai tipi wrapper. Infatti è facile dimenticare che si tratta di un **confronto tra riferimenti**, come tutti gli oggetti.

Per confrontare quindi è fondamentale utilizzare il metodo **equals** e non ==

4 Lezione del 18-03

4.1 Uguaglianza su una classe definita

```
Integer a1 =4, a2 = 4;
a1 == a2; // True

Iteger b1 = 400, b2 = 400;
b1 == b2 // False supero il range in cui l'integer si comporta come un classico int
```

Per scelta implementativo il sistema di cache (che si trova all'interno della classe stessa) si ferma ad una certa cifra. Tra tipi wrapper pertanto è fondamentale utilizzare il metodo apposito `equals` (e non == onde evitare risultati inaspettati).

4.2 Definizione della classe Integer

Per una vista completa della definizione di questa classe dai uno sguardo qui

```
// È immodificabile e inestensibile
public final class Integer extends Number{
    // Caso in cui sia a load time
    public static Integer valueOf(int n){
        if(n > 127 && n <=128){
            return cache[n+127];
        } else {
            return new Integer(n);
        }
    }

    // Definire Il 256 rappresenta il magic number: Il refactoring consigliato è assegnarlo ad una
        costante
    private static final int CACHE_SIZE = 256;
    private static final Integer[] cache = new Integer[CACHE_SIZE];
}
```

Per la sua inizializzazione abbiamo 2 scelte

- Riempire al load time la `CACHE_SIZE` mediante un blocco `static`;
- On demand;

```
static{
    for (int i = 0; i< CACHE_SIZE; i++){
        // Per lo stesso motivo di CACHE_SIZE andrebbe definito -127 all'interno di una costante
        cache[i] = new Integer(i - 127);
    }
}

// On demand la gestione viene affidata a valueOf
```

```java
// Questa versione non è thread safe -> Una soluzione è definire nella firma del metodo la keyword
    synchronized
public static Integer valueOf(int n){
    if(n > 127 && n <=128){
        // Si ha una race condition
        if(cache[n + 127] != null){
            return cache[n + 127];
        }
        else {
            return cache[n + 127] = new Integer(n);
        }
    }
}
```

4.3 Riflessione/Introspezione

Insieme di meccanismi che il linguaggio può offrire per investigare sul tipo degli oggetti a **Run-time**.

```java
/*
  La riflessione consente di scoprire il tipo effettivo di x.
  Un linguaggio staticamente tipato (come java) permette di comportarsi come un linguaggio
      dinamicamente tipato (come python)
*/
f(Object x){

}
```

A differenza di **Java**, il **C** non offre introspezione. Quello che più si avvicina al concetto di introspezione è **sizeof()** (NB. essendo statico, questo controllo avviene solo a compile time). **C++** si pone a metà (inizialmente non offriva introspezione), utilizzando la sigla RTTI (Run-Time Type Information).

In Java esiste una classe speciale chiamata **Class**. L'introspezione ruota intorno a questa classe. È un tipo di classe *managed* (gestita dal sistema).

Per ottenere questi oggetti:

- Utilizziamo l'operatore **.class** (Può essere utilizzato solo attraverso il nome della classe → ha natura statica);
- Metodo dinamico della classe Object **getClass**;

```java
// Caso 1
Employee.class;
java.lang.String.class;

// Caso 2
```

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

Employee e = new Manager(); // Manager è sottoclasse di Employee

Class<?> c = e.getClass(); // Ottengo un oggetto `manager`

// Meglio del ? c'è solo
Class<? extends Employee> c = e.getClass();
```

4.4 Uguaglianza tra oggetti

Come abbiamo detto, in Java, l'operatore == stabilisce se 2 riferimenti puntano al medesimo oggetto.

Spesso è utile considerare uguali 2 oggetti distinti, secondo criterio dettato di volta in volta dal contesto applicativo.

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

```
public boolean valueOf(Object o)
```

Ad esempio consideriamo la seguente classe:

```java
class Employee{
    private String name;
    private int salary;
    private Employee boss;
    ...
}
```

Vogliamo quindi confrontare 2 Employee. Una possibile implementazione è la seguente:

```java
// È un overriding (Sono obbligato a lasciare il parametro Object o). Se lo cambiassi avrei un
    overloading
public boolean equals(Object o){
    // Questo è un controllo ridondante (null non è instanceof Employee)
    if (o == null) return false;
    if(!(o instanceof Employee)) return false:
                                Employee e = (Employee) o;
    return name.equals(e.name) && (boss = e.boss || (boss != null && boss.equals(e.boss))):
}
```

- All'inizio, controlliamo che il riferimento passato punti effettivamente ad un `Employee`;
- Confrontiamo i nomi con `equals`;

- Confrontiamo anche i capiufficio con `equals` (sono impiegati anche loro, quindi vale per loro la stessa assunzione fatta per gli impiegati semplici);
- La forma complessa dell'espressione condizionale è dovuta al caso in cui uno dei due impiegati in questione, o entrambi, siano al vertice dell'azienda.

Il linguaggi Java richiede che qualunque ridefinizione del metodo `equals` rispetti le seguenti proprietà:

1. Riflessività → ∀ oggetto *x*, `x.equals(x)` è vero;
2. Simmetria → ∀ coppia di oggetti *(x,y)*, `x.equals(y)` è vero se e solo se `y.equals(x)` è vero;
3. Transitività → ∀ terna di oggetti *(x,y,z)* se `x.equals(y)` è vero e `y.equals(z)` è vero, allora anche `x.equals(z)` è vero.

Fondamentale è prestare attenzione all'interazione tra il metodo equals e le `sottoclassi`. Infatti, in base al contesto applicativo, in fase di progettazione, va deciso come si deve comportare il metodo `equals` con oggetti appartenenti a sottoclassi diversi.

Sorgono 2 scenari standard:

1. Il criterio di confronto è **uniforme** in tutta la gerarchia e coincide con quello che si applica alla sua radice → oggetti di sottoclassi diverse possono anche risultare uguali tra loro
    - In tutta la gerarchia vale il criterio stabilito per la sua radice;
    - Risulta in una implementazione molto semplice;
2. Il criterio di confronto **cambia** nelle sottoclassi → oggetti di sottoclassi diverse sono sempre considerati diversi
    - Solitamente, questo si traduce in un overriding del metodo in ogni classe (ridefinizione in ogni sottoclasse);

4.4.1 Caso di uguaglianza mista

I 2 scenari standard non coprono tutti i casi possibili. In particolare sono esclusi casi in cui il criterio non è uniforme, e oggetti di classi diverse possono in alcune circostanze essere considerati uguali.

Questa specifica risulta molto complessa (le specifiche di questo tipo sono sconsigliate). Si devono rispettare i seguenti principi:

- Le classi non dovrebbero essere consapevoli delle loro sottoclassi;
- Ciascuna classe dovrebbe controllare solo i propri campi

Una possibile implementazione potrebbe essere:

```java
// In Employee
public boolean equals(Object o){
    if (!(o instance Employee)){
        return false
            }
Employee e = (Employee) o;
return (name.equals(e.name) && salary ==
    e.salary);
}
```

```java
// In Manager
public boolean equals(Object o){
    if(!super.equals(o)) return false;

    if (!(o instance Manager)){
        return false;
    }
    Manager m = (Manager) o;
    return bonus = m.bonus;
}
```

Tuttavia non rispetta la specifica (realizza una relazione non simmetrica). Una implementazione corretta, ma poco pulita e leggibile è la seguente

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

```java
// In Manager
public boolean equals(Object o){
    if(!super.equals(o)) return false;

    if (!(o instance Manager)){
        return true;
    }
    Manager m = (Manager) o;
    return bonus = m.bonus;
}

public boolean isCompatibleWithEmployee()
    {
    return bonus==0;
}
```

Non conviene specializzare il tipo del parametro di `equals`, perché è meglio effettuare l'**overriding**, piuttosto che l'**overloading**.

Se avessi scelto l'overloading, i metodo dovrebbero avere la firma più permissiva possibile, compatibile con il servizio che devono offrire.

Il metodo `equals` interagisce con diverse funzionalità offerte dalla libreria standard Java. In particolare:

- Con il metodo `hashCode` di Object;
- Con le collezioni offerte dalla libreria `Java Collection Framework`.

5 Lezione del 21-03

5.1 Regole dell'overriding

Supponiamo che in una superclasse ci sia un metodo con:

- Visibilità V;
- Tipo T;
- Lista di parametri;
- Serie possibile di Eccezioni E;

In una sua possibile sottoclasse:

- La sua visibilità può solo aumentare

Visibilità (superclasse)	Visibilità (sottoclasse)
public	public
private	Non è possibile eseguirne l'overriding
default (solo nel package)	default, protected, public
protected (package e sottoclassi)	protected, public

In una sua possibile sottoclasse:

- Il suo tipo può cambiare (in precedenza non era così) → il tipo può restare lo stesso o diventare suo sottotipo (se T è primitivo non può cambiare; se T è assegnabile può diventare sottotipo).

In una sua possibile sottoclasse:

- I parametri formali non possono variare;

In una sua possibile sottoclasse:

- Le eccezioni (molto probabilmente checked, che il metodo potrebbe sollevare) possono variare numericamente (possono diminuire e aumentare). Fondamentale è che ognuna delle eccezioni presenti nel metodo della sottoclasse deve essere sottotipo di uno di quelle presenti nel metodo della superclasse.

$$\forall\, i = 1 \dots n \,\exists\, j = 1 \dots m \,:\, F_i \text{ sottotipo di } E_j$$

5.2 Classi interne, locali e anonime

Java permette di definire una classe (o interfaccia) all'interno di un altra.

Tale meccanismo permette di aumentare le possibilità di relazioni tra classi, introducendo nuove regole di visibilità.

Se la definizione di una classe si trova all'interno di un metodo, tale classe prende il nome di **locale**.

Una classe locale può essere anche **anonima**, quando il suo nome non sarebbe rilevante `e/o` utile.

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

1. **Privilegi di visibilità** rispetto alla classe Top level e altri classi in essa contenute → Permettono una stretta collaborazione tra queste classi;
2. **Restrizioni di visibilità** rispetto alle classi esterne a quella Top level → Permettono di nascondere la classe all'interno (incapsulamento);
3. Un **riferimento implicito** ad un oggetto della classe Top level → Ogni oggetto della classe interna "conosce" l'oggetto della classe contenitrice che l'ha creato.

Oltre a campi e metodi, una classe può contenere altre classi o interfacce, dette *interne*. Queste classi possono avere tutte le 4 visibilità ammesse dal linguaggio.

Tra classi contenute nella stessa classe non vige alcuna restrizione di visibilità

Ciascun oggetto di una classe interna (non statica) possiede un riferimento implicito ad un oggetto della classe Top level. Questo riferimento:

- Viene inizializzato automaticamente al momento della creazione dell'oggetto;
- Non può essere modificato.

Le classi interne possono essere statiche o meno.

Una classe interna dichiarata nello scope di classe (cioè al di fuori di metodi e inizializzatori) è statica se preceduta dal modificatore `static`.

Una classe interna dichiarata all'interno di un metodo (quindi locale) o di un inizializzazione eredita il proprio essere statica o meno dal metodo in cui è contenuta o dal campo che si sta inizializzando.

Le classi interne statiche non possiedono il riferimento implicito alla classe contenitrice.

Le classi statiche godono solo delle prime 2 proprietà precedentemente elencate.

es.

```java
public class InternalVisibility{
    private int n = 42;
    public static class A {
        public void foot(InternalVisibility guest){

            // Se la classe A fosse top level non sarebbe possibile accedere al campo n
            guest.n = 0;
        }
    }

    public static class B extends InternalVisibility {
        public void foot(){
            // Sintassi non valida (Errore a tempo di compilazione)
            n = 0;

            // Sintassi non valida (Errore a tempo di compilazione)
            this.n = 0;

            // Definizione legale
            // Se la classe B fosse top level non sarebbe possibile accedere al campo n
            super.n = 0;
        }
    }

}
```

Una classe interna dichiarata all'interno di un metodo viene detta `locale`.

Questo tipo di classe non ha specifica di visibilità, in quanto è visibile solo all'interno del metodo in cui è dichiarata.

Una classe `locale` non può avere modificatore static, in quanto eredita il suo essere statica o meno dal metodo in cui è dichiarata.

Godono delle proprietà comuni alle classi interne; inoltre hanno la possibilità di *vedere* le variabili locali e i parametri formali del metodo in cui sono contenute, a patto che siano `effectively final` (Di base sono tutte final fin tanto che non vengano assegnate).

Le istanze di una classe locale possono vivere più a lungo del metodo in cui la loro classe è visibile.

Tipicamente, questo succede quando un oggetto di una classe locale viene restituito dal metodo, mascherato da una superclasse o super-interfaccia nota all'esterno.

In questi casi, l'accesso alle variabili locali (compresi i parametri formali) di quel metodo può avvenire quando quel metodo è ormai terminato, cancellando di fatto quelle variabili (si ricordi che le variabili locali e i parametri formali sono allocati sullo stack

e quindi vengono cancellati al termine di ciascuna invocazione del metodo)

Per dare l'illusione al programmatore di accedere a quelle variabili, il compilatore inserisce negli oggetti delle classi locali delle copie di quelle variabili.

6 Lezione del 25-03

Per le esercitazioni il prof farà utilizzo della piattaforma Crowdgrader.

6.1 Polimorfismo parametrico - I Generics

È la possibilità di dotare classi, interfacce e metodi di parametri di tipo.

Simili ai normali parametri dei metodi, questi parametri hanno come possibili valori i tipi (non primitivi) del linguaggio.

Questo meccanismo consente di scrivere codice più robusto dal punto di vista dei tipi di dati, evitando in molti casi il ricorso alle conversioni forzate.

Risultano molto utili nella realizzazione di collezioni, classi deputate a contenere altri oggetti.

Supponiamo di voler utilizzare una classe, chiamata `Pair`, che rappresenta una coppia di oggetti dello stesso tipo. In mancanza della programmazione parametrica la classe si sarebbe dovuta realizzare in questo modo:

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

Una sua implementazione obbligherebbe a utilizzare un cast, perché gli elementi estratti dalla coppia riacquistino il loro tipo originario, come il seguente esempio:

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

La maniera corretta, rendendo la classe parametrica è la seguente:

```
class Pair<T>{
    private T first, second;
    public Pair(T a, T b){
        first = a;
        second = b;
    }
```

```
    public T getFirst(){ return first; }
    public void setFirst(T a ){ first = a; }
}
```

In questo caso, la classe `Pair` ha un parametro di tipo, chiamato `T`. I parametri di tipo vanno dichiarati dopo il nome della classe, racchiusi tra parentesi angolari.

Se vengono dichiarati più parametri di tipo, questi vanno separati da virgole.

All'interno della classe, un parametro di tipo si comporta come un **tipo di dati vero e proprio**, tranne che per alcune eccezioni.

In particolare, un parametro di tipo può usare come tipo di un campo, tipo di un parametro formale di un metodo e tipo di ritorno di un metodo;

La nuova versione di Pair permette agli utenti della classe di specificare di che tipo di coppia si tratta e, così facendo, di evitare i cast:

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

La sintassi <> prende il nome di `operatore diamond`

Nella dichiarazione della variabile `p` è obbligatorio indicare il parametro attuale di tipo.

Nell'istanziazione dell'oggetto Pair, tale indicazione è facoltativa (in questo contesto).

```
Pair<Employee> p = new Pair<Employee>(..., ...);
Employee a = p.getFirst();
```

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

Per mantenere compatibilità con vecchie versioni di Java, è possibile usare una classe parametrica come se non lo fosse.

Quando utilizziamo una classe parametrica senza specificare i parametri di tipo, si dice che stiamo usando la versione *grezza* di quella classe.

Le classi grezze esistono solo per retro-compatibilità → Il codice nuovo **dovrebbe sempre specificare** i parametri di tipo delle classi parametriche.

6.1.1 Versione particolare di Pair con 2 oggetti di tipo diverse

In questo caso la classe avrà 2 parametri di tipo, che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia:

```
class Pair<T, U>{
```

```
    private T first;
    private U second;
    public Pair(T a , U b){
        first = a;
        second = b;
    }
    private T getFirst() { return first; }
    private U getSecond() { return second; }

    private void setFirst(T a){ first = a; }
    private void setSecond(U b){second = b;}
}

// Una sua dichiarazione
Pair<String, Employee> p = new Pair<>("Pippo", new Employee(...));
```

6.1.2 Esercizio

Specifica:

1. Creazione di una sequenza vuota;
2. Inserimento alla fine;
3. Iterazione;

```
Sequence<Integer> s = new Sequence<>();
s.insert(3);

// Prima soluzione
// Presenta un problema di complessità (si veda il caso in cui si voglia implementare la sequence
    come lista concatenata)
for (int i = 0; i< s.lenght(); i++){
    System.out.println(s.get(i));
}

// Seconda soluzione (approccio funzionale)
s.forEach(Consumer<Integer>)

// Terza soluzione
// Questa soluzione implica impedisce di scorrere la lista più di una volta (può essere risolto
    creando una funzione di restart)
// Il problema importante è nel caso in cui si volesse utilizzare in forma multithreading
Integer n;
while((n = s.getNext()) != null){
    System.out.println(n);
```

```
}

// Quarta soluzione (Iteratori). Il gestore della sequenza non è la sequenza stessa, ma l'
    itineratore.
Iterator<Pair> i = s.iterator();
while (i.hasNext()){
    System.out.println(i.next());
}
```

Analizzando nel dettaglio la quarta possibile soluzione (immaginando che sia di tipo `lista linkata`):

```
Interface Iterator<T>{
    boolena hasNext();
    T next();
    ...
        }

public class Sequence<T>{
    // Per il concetto di nodo (necessario alla lista linkata) definiamo il concetto di nodo (una
        classe) come classe interna (Lest privilage)
    // La scelta di un parametro di tipo della classe Node dipende dalla scelta di definire la classe
         static o no
    private class Node{
        // Essendo la classe private è inutile definire gli attributi private
        T val;
        Node next;
        Node(T val, Node next){ ... }
    }

    // Una sequence ha almeno un nodo (la testa) e una coda (l'inserimento è in coda)
    private Node head, tail;

    public void insert(T val){
        if (head == null){
            head = tail = new Node(val, null);
        } else{
            tail.next = new Node(val, null);
            tail = tail.next;
        }
    }

    public Iterator<T> iterator(){
        return new Iterator<>(){
            Node cur = head;
            @Override
            public boolean hasNext(){
```

```
            return cur != null
        }

        @Override
        public boolean next(){
            T result = cur.val;
            cur = cur.next;
            return result;
        }
    }
  }
}
```

6.2 Java collection Framework

Parte della libreria standard che contiene collezioni e suoi algoritmi.

Ne fanno parte le `List`, interfaccia che rappresenta una sequenza di oggetti, che presenta due classi concrete:

- `ArrayList();`
- `LinkedList();`

I metodi contenuti in List sono i seguenti:

- `int size();`
- `boolean add(T)` → Il metodo restituisce `true` nel caso in cui sia andato a buon fine l'inserimento;
- `boolean contains(Object)` → Il metodo restituisce `true` se la lista contiene un oggetto uguale (secondo equals) a quello passato (si passa Object per essere più ampio);
- `boolean remove(Object)` → Il metodo restituisce `true` nel caso in cui sia andato a buon fine la rimozione (rimuove il primo elemento che combacia con l'oggetto passato (ricordiamo che a differenza dei `Set` le liste ammettono ripetizioni));
- `Iterator<T> iterator();`

Complessità dei metodi precedenti:

	size	add	contains	Remove	Iterator
ArrayList	costante	Dipende dal resize (lineare)	lineare	lineare	Costante
LinkedList	costante	costante	lineare	lineare	Costante

7 Lezione del 28-03

7.1 Enhanced-for (for-each)

```
for(T x: exp){
    ...
}
```

È valido se:

- `exp` deve implementare (direttamente o non) il tipo `Iterable` → A tempo di compilazione viene trasformato in:

```
Iterator<t> i = exp.iterator();
while(i.hasNext){
    T x = i.next;
}
```

- Il tipo dichiarato di `exp` è un array o è un sottotipo di `T` → A tempo di compilazione viene trasformato in un classico ciclo `for`;

In `Iterator<T>` è presente un terzo metodo:

```
// Ha il compito di rimuovere l'ultimo oggetto restituito da next da una collezione
// Risulta utile per filtrare una collezione
default void remove(){
    throw new UnsupportedOperationException("remove");
}
```

Dove `default` è una keyword che indica che il metodo presenta un corpo. Ciò permette di non dover implementare i metodi nelle classi che implementano questa interfaccia.

7.2 Programmazione tramite contratti

L'idea consiste nell'applicare al Software, in particolare a quello orientato agli oggetti, la nozione comune di *contratto*.

È quindi un accordo 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 richieste.

Dal punto di vista del chiamante:

- Il compito svolto dal metodo è un beneficio;
- Le pre-condizioni sono obblighi.

Dal punto di vista del metodo:

- Il compito da svolgere è un obbligo;
- Le pre-condizioni sono un beneficio (agevolano o consentono il compito).

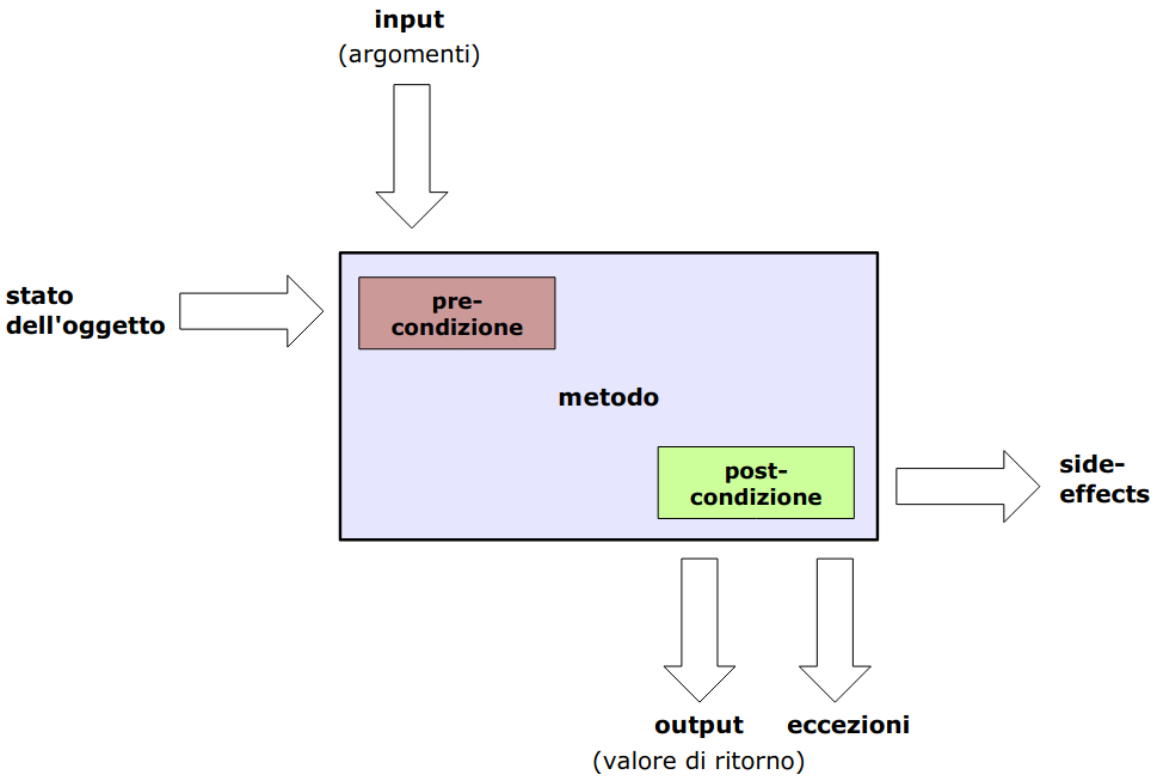
La pre-condizione riguarda:

- I valori passati al metodo come argomenti;
- Lo stato dell'oggetto su cui viene invocato il metodo.

Il metodo che assolve, detto anche **post-condizione** riguarda:

- Valore restituito;
- Il nuovo stato dell'oggetto;
- Qualsiasi altro effetto collaterale;

NB: Tutto quello che fa un metodo e che è visibile all'esterno, eccetto restituire un valore, è considerato un effetto collaterale (*side effect*).



Inoltre il contratto può specificare come reagisce il metodo nel caso in cui le pre-condizioni non siano soddisfatte dal chiamante → In Java, questa penale, consiste solitamente nel lancio di una eccezione non verificata.

La pre-condizione descrive l'uso corretto del metodo → La sua violazione costituisce **errore di programmazione da parte del chiamante**.

La post-condizione descrive l'effetto atteso dal metodo → La sua violazione indica un **errore di programmazione nel metodo stesso**.

In qualsiasi linguaggio di programmazione, solo una parte del contratto può essere espressa nel linguaggio stesso (il resto sarà indicato nel linguaggio naturale).

Il nome e tipo dei **parametri formali** di un metodo fanno parte delle pre-condizioni;

Il **tipo di ritorno** fa parte della post-condizione.

La dichiarazione di `throws` andrebbe utilizzato solo per le eccezioni `verificate`, mentre la violazione di una **pre-condizione** richiede una eccezione non verificata (si tratta di un errore da parte del programmatore).

Questa clausola, usata correttamente con eccezioni verificate, esprime parte della post-condizione (è compito suo quello di descrivere le eventuali condizioni anomale che possano dare luogo ad eccezioni verificate).

In un contratto ben definito, la pre-condizione contiene tutte le proprietà che servono al metodo di svolgere il suo compito.

7.2.1 Contratto dei metodi dell'interfaccia `Iterator<T>`

Il contratto, nelle interfacce, risulta molto importante, perché rappresenta la sua vera *raison d'etre*.

Metodo `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.

Metodo `next`:

- Pre-condizione → Ci sono ancora elementi su cui iterare (un'invocazione precedente di `hasNext` restituisce `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.

Metodo `remove`:

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

In particolare, il metodo viene indicato come opzionale:

- Ciò significa che le implementazioni di `Iterator` non sono obbligate a supportare questa funzionalità;
- Se un implementazione non vuole supportarla →
  - Deve far lanciare al metodo remove l'eccezione `UnsupportedOperationException()` (non verificata).

7.3 Contratti ed overriding

Il contratto di un metodo andrebbe considerato vincolante anche per le eventuali ridefinizioni dei metodi.

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

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

Quindi, ogni ridefinizione del contratto dovrebbe offrire al client almeno gli stessi benefici, richiedono al più gli stessi obblighi.

In altre parole, un overriding può:

- Rafforzare la post-condizione (garantire di più);
- 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.

Le regole di overriding in Java rispecchiano solo una metà delle regole contro-variante:

- Il tipo di ritorno può diventare più specifico nell'overriding, rafforzando quindi la post-condizione;
- Il tipo dei parametri non può cambiare, mentre la regola contro-variante prevederebbe che potessero diventare più generali.

7.4 Javadoc

Tool che estrae documentazione dai sorgenti Java e la rende disponibili in vari formati.

Estrae informazioni dalle dichiarazioni e da alcuni commenti speciali (racchiusi tra `/** e */`). All'interno di questi commenti, si possono utilizzare dei tag per strutturare la documentazione.

Il contratto di un metodo andrebbe indicato in un commento che precede il metodo, utilizzando almeno i tag:

`@param`, `@return` e `@throws`

es

```
1
2  /**
3   *
4   * @param x numero non-negativo di cui si vuole calcolare la radice quadrata
```

```
 * @return la radice quadrata di x
 * @throws IllegalArgumentException se x è negativo
*/
public double sqrt(double x){
    if (x < 0) throw new IllegalArgumentException();
}
```

7.5 Parti del contratto

In alcuni casi, è conveniente distinguere 2 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.

Ad esempio, il metodo `equals` è destinato ad essere ridefinito nelle sottoclassi; il contratto è diviso in parte *generale* e in parte *locale* (che descrive il comportamento della versione originale presente in Object, ma non è vincolante per le sottoclassi di Object).

7.5.1 Parte generale

- **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 `true` 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*).

7.5.2 Parte locale

- **Pre-condizione** → Nessuna;
- **Post-condizione** → L'invocazione `x.equals(y)` è equivalente a *x == y* (quando x non sia `null`).

8 Lezione del 01-04

8.1 Contratti nel linguaggio Effeil

```
class
    ACCOUNT

feature
```

```
  balance: Integer
    -- Current balance
  deposit_count: INTEGER

feature
  deposit (sum: INTEGER)
    -- Add `sum` to account
  require
    non_negative: sum >= 0 (Named precondition)
  do
    ... metod body ...
  ensure
    one_more_deposit: deposit_count = old deposit_count + 1 (Named postcondition)
    updated: balance = old balance + sum
  end
```

Queste asserzioni vengono controllate a runtime, se attivate a tempo di compilazione.

Per sfruttare al meglio il *design by contract*:

- Documentare pre e post condizione;
- Mantenere una disciplina progettuale;
- Scegliere quando monitorare le condizioni (a runtime o a tempo di compilazione) → In un linguaggio di Java si punta a controllarle a runtime;

Nel caso in cui si voglia gestire una post-condizione è possibile fare uso delle `assertion`. Prendendo l'esempio utilizzato per la Javadoc:

```
double result = Math.sqrt(x);

// NB: Per utilizzare le assertion va abilitato in fase di compilazione (con enable assertion)
assert(Math.abs(result * result -x ) <= EPS);

return result;
```

8.2 Esercitazione su metodi

Supponiamo di scrivere un metodo che accetti un array come argomento e ne stampi il suo contenuto:

```
void printAll(Object[] a){
   for (Object x: a){
      System.out.println(x)
   }
}
```

Supponiamo adesso di volerlo fare con le liste:

```
void printAll(List<Object> l){

    // Questa versione non funziona con altri tipi di Liste: Una List<Object> non è supertipo di
        altri tipi di liste
    // Le classi parametriche non preservano la relazione di sottotipo
    for (Object x: l){
        System.out.println(x)
    }
}
```

Questo significa che non è permesso fare:

```
List<Object> l = new ArrayList<String>();

Arraylist<Object> l = new ArrayList<String>();
```

È presente questa restrizione per ridurre dei possibili errori a runtime. Per ovviare a questo problema si utilizzano dei metodi parametrici:

```
// Dichiarazione di un parametro di tipo locale a questo metodo
<T> void printAll(List<T> l){
    for (T x: l){
        System.out.println(x)
    }
}
```

Vogliamo scrivere un metodo che accetti un array e ne restituisca l'elemento centrale:

```
public class Test{
    public static <T> T getMedian(T[] a){
        return a[a.lenght / 2];
    }
}
```

Per invocarli indicare il valore di tipo è opzionale:

```
Employee[] array = new Employee();

// Questa è la versione consigliata
Employee e = Test.<Employee>getMedian(array);

// Ma è possibile farlo anche senza la notazione diamond
// In questo caso utilizziamo la type inference (chiediamo al compilatore di dedurre il parametro
    attuale di tipo)
```

```
// In questo modo però non c'è garanzia di successo certo
Employee e = Test.getMedian(array);
```

Scriviamo un metodo che abbia lo scopo di riempire tutte le celle di un array con un oggetto:

```
// Questo metodo può creare problemi nel caso in cui venga utilizzata la type inference
public class Test{
    ...

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

    ...

    // Usiamo la type inference
    Test.fill(array, new Employee(...));

    // Questo errore non viene riscontrato: T viene associato a Object (che è una cosa sbagliata)
    // In compilazione non ha problemi, ma a runtime solleva un'eccezione -> L'array conserva il suo
        oggetto originale (che in questo caso non è un Integer)
    // Solleva ArrayStoreException
    Test.fill(array, new Integer(7));

    // Per questo motivo si preferisce utilizzare una forma esplicita
    // In questo modo il codice non compila
    Test<Integer>.fill(array, new Integer(7));
}
```

```
class A<T>{
    void f(int n){
        ...
    }

    // Metodo parametrico che vede T ed ha un parametro locale `U` (metodo istanza)
    <U> void g(...){...}
}
```

8.3 Confronto tra oggetti

La libreria standard Java fornisce 2 interfacce per l'ordinamento per gli oggetti di una classe

8.3.1 Interfaccia Comparable

```
public interface Comparable<T>{
    // Pre condizione: L'oggetto x deve essere confrontabile con this
    /* Post condizione: restituisce:
      + Valore negativo se this è minore di x;
      + 0 se this è uguale a x;
      + Un valore positivo se this è maggiore di x;

    */
    public int compareTo(T x);
}
```

Il metodo dovrebbe lanciare l'eccezione (non verificata) *ClassCastException* se riceve un oggetto che, a causa del suo tipo effettivo, non è confrontabile con this.

L'uso di questa interfaccia è indicata quando la classe da ordinare possiede un unico criterio di ordinamento naturale.

La scelta di utilizzare un intero è analogo alla funzione `strcmp` del linguaggio C (che confronta alfabeticamente le stringhe puntate da s1 ed s2, restituendo un valore intero secondo le stesse regole del contratto di `compareTo`).

es:

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

    @Override
    public int compareTo(Employee x){
        return name.compareTo(x.name);
    }
}
```

In alternativa si può utilizzare una seconda classe che implementi l'interfaccia `Comparator`

8.3.2 Interfaccia Comparator

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

Il contratto del metodo `compare` di Comparator è analogo a quello di `compareTo` di Comparable.

È indicato quando:

- La classe da ordinare non ha un unico criterio di ordinamento naturale;

- La classe da ordinare è già stata realizzata e non si può o non si vuole modificarla

es:

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

    // Si noti l'uso di una classe anonima per l'inizializzazione di un campo statico.
    // Si potrebbe in alternativa utilizzare una lambda expression
    public static final Comparator<Employee> comparatorByName = new Comparator<>(){
           public int compare(T a, T b){
               return a.name.compareTo(b.name);
           }
       };

    public static final Comparator<Employee> comparatorBySalary = new Comparator<>(){
           ...
       };
}
```

Le stringhe sono dotate di un ordinamento naturale che è quello alfabetico (o *lessicografico*).

La classe String fornisce questo criterio di confronto implementando Comparable.

Può essere utile anche ordinare stringhe senza considerare la distinzione tra minuscole e maiuscole (*case insensitive*). È presente per questo motivo la seguente costante:

public static final Comparator<String> `CASE_INSENSITVE_ORDER`;

La classe string offre un criterio di confronto naturale (`compareTo`) e un criterio alternativo, sotto forma di oggetto Comparator disponibile ai client tramite una costante di classe.

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

- Dato un numero reale `a`, se definiamo la seguente funzione *segno* `sgn(a)`:

$$sgn(a) = \begin{cases} 1 \; se \; a \; > \; 0 \\ 0 \; se \; a \; = \; 0 \\ -1 \; se \; a \; < \; 0 \end{cases}$$

- Dati tre oggetti `x`, `y`, e `z`, appartenenti ad una classe che implementa Comparable, deve valere:
  - `sgn(x.compareTo(y)) == - sgn(y.compareTo(x))`;
  - Se `x.compareTo(y) < 0` e `y.compareTo(z) < 0` allora `x.compareTo(z) < 0`;
  - Se `x.compareTo(y) = 0` allora `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`;

Condizioni analoghe devono valere per le implementazioni di Comparator.

È indicato che:

- Le classi che implementano Comparator siano `stateless`.
- Le classi che implementano Comparable siano `stateful`;

Inoltre è preferibile (non obbligatorio), che le implementazioni di Comparable e Comparator siano coerenti con *equals*.

8.3.3 Uso di comparatori per ordinare array

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

Per i primi tali metodi si trovano nella classe **java.util.Arrays**, una classe contenente solo metodi statici:

```
1 // Ordina l'array a in senso non-decrescente, in base all'ordinamento naturale tra i suoi elementi
2 // Suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia
      Comparable
3 public static void sort(Object[] a);
4
5 // Ordina l'array in senso non-decrescente, in base all'ordinamento indotto dal comparatore c
6 public static <T> void sort(T[] a, Comparator<T> c); // versione semplificata
```

In entrambi i casi, l'ordinamento è *in-place* e *stabile*:

- L'array viene modificato senza utilizzare strutture di appoggio e gli elementi equivalenti secondo l'ordinamento mantengono l'ordine che avevano originariamente.

L'algoritmo usato è una versione ottimizzata del *merge sort*.

8.3.4 Uso di comparatori per ordinare liste

I metodi di ordinamento si trovano nella classe **java.util.Collections**, una classe che contiene solo metodi statici

```
1 // Ordina l'array a in senso non-decrescente, in base all'ordinamento naturale tra i suoi elementi
```

```java
// Suppone che tutti gli elementi contenuti siano confrontabili tra loro tramite l'interfaccia
    Comparable
public static void sort(List l);

// Ordina l'array in senso non-decrescente, in base all'ordinamento indotto dal comparatore c
public static void sort(List l, Comparator c);
```

In entrambi i casi, l'ordinamento è *in-place* e *stabile*. L'algoritmo usato è una versione ottimizzata del *merge sort*.

NB: Per semplicità, sono state presentate le versioni *grezze* dei metodi

8.3.5 Confronto con l'ordinamento in C

È interessante confrontare i metodi di sort di Java con quello presente nella libreria standard del linguaggio C

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

Dove:

- I primi 3 argomenti servono a passare un array di tipo arbitrario, specificandone l'indirizzo di base, la lunghezza e la dimensione di ciascuna cella (questo per la scelta del linguaggio di non avere overhead → A runtime gli array non conoscono le proprietà necessarie al metodo per eseguire il sorting);
- L'ultimo argomento è un puntatore a una funzione che accetta 2 elementi generici da confrontare (`void *`) e restituisce un intero.

L'analogia con `Comparator` è evidente → La funzione `qsort` è storicamente precedente al linguaggio Java.

9 Lezione del 04-04

9.1 Parametri di tipo con limiti superiori

```java
// Questo metodo non accetta Liste di Sottotipi di Employee
int totalSalary(List<Employee> l){
    int tot = 0;
    for (Employee e: l){
        tot += e.getSalary();
    }
    return tot;
}

// Il metodo quindi diventa

// Dichiaro un nuovo parametro di tipo che presenta come limite superiore la classe Employee
```

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

Nel secondo metodo posso utilizzare il metodo per tutti i sottotipi di `Employee`:

```
List<Manager> l;
int tot = Test.<Manager> totalSalary(l);
```

9.1.1 Regole

Ogni qual volta si dichiara un nuovo parametro di tipo (che sia di una classe o di un metodo), può avere uno o più limiti superiori. La sintassi è la seguente:

```
<id extends L1 & L2 & L3 & ... & Ln> ...
```

In questo elenco solo `L1` può essere una classe. Gli altri eventuali limiti devono essere interfacce (nonostante si utilizzi la keyword `extends`).

Errore comune è definire il limite nell'uso e non nella dichiarazione:

```
// Corretto
<T extends Employee> int totalSalary(List<T> l)...

// Errato
<T> int totalSalary(List<T extends Employee> l)...
```

9.2 Parametro Jolly ?

Vogliamo realizzare un metodo che accetti una Lista che prenda il minimo di oggetto confrontabili con `Comparable`

```
<T extends Comparable<T>> T getMin(List<T> l)
```

Il problema di questa definizione, è la perdita del sottotipo → Non viene rispettato il limite superiore

Per ovviare a questo problema introduciamo il parametro `Wildcard/Jolly ?`, un parametro `attuale` di tipo:

```
// ? è supertipo comune di tutte le classi interfacce-classi parametriche
// È il modo giusto di generalizzare (Ricordiamo che List<Object> non è supertipo degli altri)
List<?> = new LinkedList<String>();
```

In questo modo il metodo `printAll(List<T>)` diventa:

```
void printAll(List<?> l){
// Non essendo valida una cosa del tipo `? o : l` Utilizzo Object
    for(Object o: l){
    ...
    }
}
```

Potendo scegliere tra `T` e `?` conviene sempre utilizzare `?`

```
// Accetta 2 liste qualsiasi
f(List<?> l1, List<?> l2);

// Accetta 2 liste dello stesso tipo
f(List<T> l1, List<T> l2);
```

9.2.1 Limiti applicati a ?

Sintassi:

```
// Assegnazioni valide
List<? extends Employee> l = new ArrayList<Employee>()
List<? extends Employee> l = new ArrayList<Manager>()

// Assegnazioni non valide
List<? extends Employee> l = new ArrayList<String>()
```

Il metodo `totalSalary` applicato a `?` diventa:

```
// Può essere iterato anche ai suoi supertipi (non ai suoi sottotipi)
// Continuo a non poter chiamare il metodo add di l (potrei inserire soltanto `null`)
int totalSalary(List<? extends Employee> l){
    int tot = 0;

    // Andiamo a iterare con Employee invece di Object
    for (Employee e: l){
        tot += e.getSalary();
    }
    return tot;
}
```

Viene mantenuto lo stesso corpo precedente (non è più parametrico)

9.2.2 Limiti inferiori di ?

Sintassi:

```
// Posso assegnare a l qualsiasi supertipo di Employee
List<? super Employee> l = new ArrayList<Employee>();
List<? super Employee> l = new ArrayList<Person>();
List<? super Employee> l = new ArrayList<Object>();

// Non valida
List<? super Employee> l = new ArrayList<Manager>();
```

In questo caso si risolve il problema del metodo `add` (è valido infatti inserire un nuovo `Employee`)

Accettando un `Employee` accetterà anche un suo sottotipo (accetta infatti un `Manager`).

In questo contesto `add` accetta `Employee` e suo sottotipi.

Continua a non essere safe aggiungere una `Person`.

NB: È possibile settare un limite solo in un senso (o `extends` o `super`)

Per completare il metodo `getMin`:

```
<T extends Comparable<? super T>> T getMin(List<T> l)
```

10 Lezione del 08-04

10.1 Esercitazione

Il seguente metodo estrae la testa di una LinkedList e la inserisce in coda:

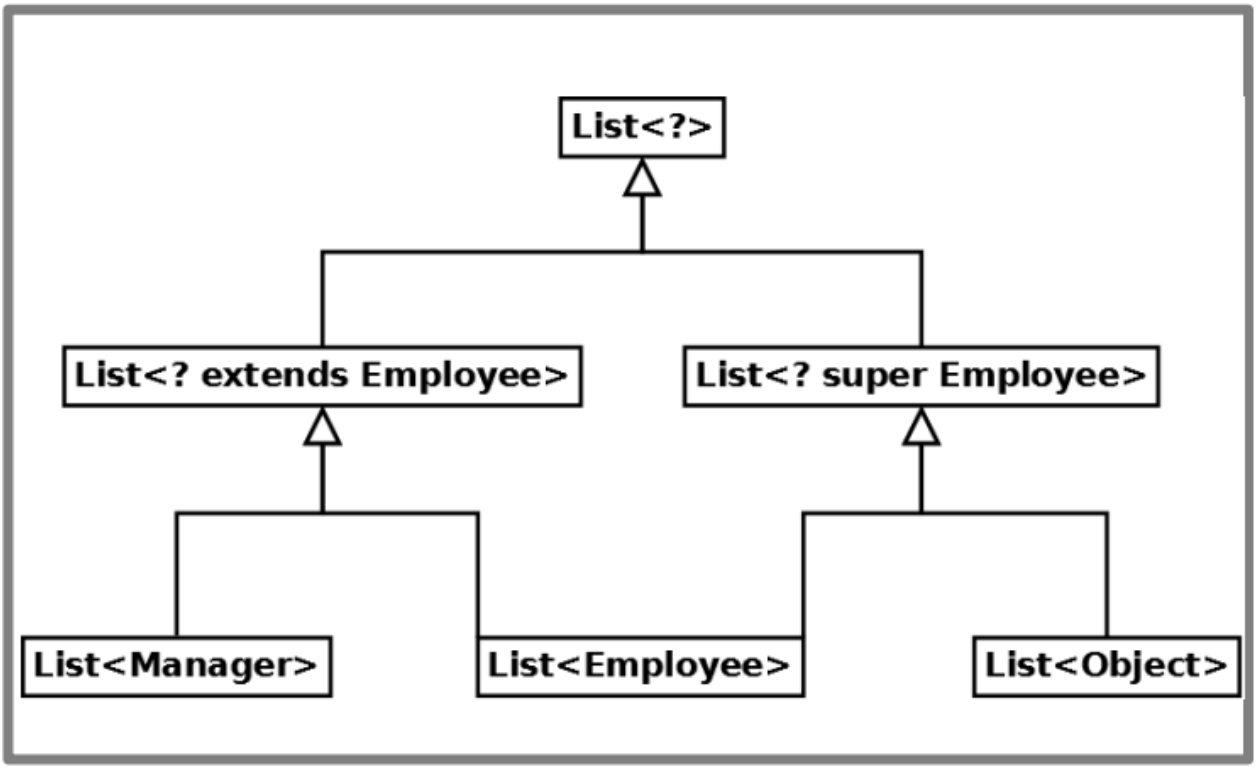
```
public static <T> void moveHeadToTail(LinkedList<T> l){
    T head = l.removeFirst();
    l.addLast();
}
```

Se sostituisco `T` con `?` non possiamo svolgere lo stesso compito:

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

10.2 Parametro di tipo Jolly con limiti superiori e inferiori

Come i normali parametri di tipo, anche il parametro di tipo Jolly può avere un limite superiore (è unico).



Ad es nel caso rappresentato `List <? extends Employee>` è una lista di tipo sconosciuto che estende Employee (è il supertipo comune a tutte le liste il cui parametro di tipo estende `Employee`).

A differenza dei normali parametri di tipo, il tipo jolly può avere anche un **limite inferiore**.

Ad es. `List<? super Employee>` rappresenta una lista di tipo sconosciuto che è supertipo di `Employee` (come Person o Object).

10.3 Esercizio

Realizzare un metodo che accetta una collezione di valori numerici e ne restituisce la somma.

Le collezioni non possono contenere tipo primitivi, quindi ci riferiamo alle classi wrapper di tipo numerico.

Tutte queste classi estendono `Number` (che offre il metodo `doubleValue`, che converte in double questo valore numerico, qualunque sia il suo tipo):

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

```
    double sum = 0.0;
    for (Number n: c){
        sum += n.doubleValue();
    }
    return sum;
}
```

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

```
// Per accettare qualsiasi collezione dotata di ordinamento naturale dobbiamo utilizzare:
public static <T extends Comparable<? super T>> void getMinMax(Collection<T> c, Pair<? super T> p) {
    T min = null, max = null;
    for (T x: c) {
        if (min==null || x.compareTo(min)<0)
            min = x;
        if (max==null || x.compareTo(max)>0)
            max = x;
    }
    p.setFirst(min);
    p.setSecond(max);
}
```

Il primo argomento potrebbe essere anche dichiarato di tipo `Collection<? extends T>` per esprimere la garanzia che la collezione non verrà modificata.

L'uso del parametro jolly con limiti superiori o inferiori impone determinate condizioni sulle chiamate ai metodi:

La seguente tabella riassume le limitazioni che valgono per un riferimento di tipo `A<?>`, `A<? extends B>`, oppure `A<? super B>`, rispetto ad un metodo di `A` che accetta un argomento di tipo `T`, oppure restituisce un valore di tipo `T`

Tipo di riferimento	Cosa si può passare a `f(T)`	A cosa si può assegnare `T f()`
`A<?>`	Solo `null`	Solo ad `Object`
`A<? extends B>`	Solo `null`	a `B` e suoi supertipi
`A<? super B>`	`B` e suoi sottotipi	Solo ad `Object`

10.4 Implementare i Generics

La sintassi per le classi e i metodi `template` in `C++` è simile a quella di Java, ma presentano l'implementazione in maniera molto diversa:

```
template<class T1, clas T2>
struct pair{
```

```
  T1 first;
  T2 second;

  pair(const T1& a, const T2& b): first(a), second(b){

  }
}
```

Quando il compilatore C++ trova un riferimento ad una versione concreta di un template come `pair<string, employee>`, esso istanzia una nuova copia della classe pair, con string al post di `T1` ed employee al posto di `T2`.

Questo approccio, detto *reificazione a tempo di compilazione* è diametralmente opposto a quello di Java.

Riassumendo:

- C++ → **Reificazione** a tempo di compilazione
  - Il compilatore crea una versione specifica del codice per ogni parametro attuale di tipo;
- C# → **Reificazione** a tempo di esecuzione
  - Come in C++, ma a tempo di esecuzione (on demand);
- Java → **Cancellazione** (erasure)
  - Il compilatore usa i Generics per fare un type checking più accurato, poi scarta l'informazione.

Il principi di funzionamento per Java è il seguente:

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

Di conseguenza, nel **bytecode** risultato della compilazione **non c'è più traccia dei parametri di tipo** → In fase di esecuzione i tipi parametrici sono scomparsi (Fanno eccezione alcune funzionalità di riflessione, che sono in grado di recuperare a run-time alcuni parametri di tipo specificati nel sorgente).

10.5 Limitazioni dei Generics

1. Non è possibile utilizzare un parametro di tipo per istanziare oggetti;
2. Non è possibile istanziare un array di tipo parametrico;

3. Non è possibile usare un parametro di tipo per distinguere 2 versioni di un metodo;
4. Dopo l'erasure, un metodo parametrico potrebbe andare in conflitto con uno non parametrico;
5. Non è possibile utilizzare un parametro di tipo per selezionare una determinata versione di un metodo in overloading;
6. I parametri di tipo non vanno usati per effettuare conversioni esplicite;
7. Non si può applicare `istanceof` a un parametro di tipo o a una classe parametrica;

10.6 Vantaggi della reificazione e della cancellazione

Reificazione (`C++` e `C#`)	Cancellazione (Java)
Espressività: Si può fare con un parametro di tipo tutto quello che si può fare con un tipo concreto	Evita il code bloating (ripetizione, nell'eseguibile o in memoria, di codice simile)
	Supporta la compilazione separata

11 Lezione del 11-04

11.1 Java Collection Framework - Collezioni

Il JCF è una parte della libreria standard dedicata alle collezioni, intese come classi deputate a contenere altri oggetti.

Questa libreria offre strutture dati di supporto tra, come liste, array (di dimensioni dinamiche), insiemi, mappe associative (dette anche *dizionari*) e code.

Le classi e interfacce del JCF si dividono in:

- Collection;
- Map

La classe **Collections** contiene numerosi algoritmi di supporto (ad *es.* metodi che effettuano l'ordinamento).

L'interfaccia Collection estende la versione parametrica di Iterable

11.1.1 Iterator e Iterable

2 interfacce parametriche:

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

```

```
7 public interface Iterable<E>{
8     public Iterator<E> iterator();
9 }
```

Lo scopo ultimo del parametro di tipo consiste nel permettere al metodo next di restituire un oggetto del tipo appropriato, evitando che il chiamante debba ricorrere ad un cast

11.1.2 List

Rappresenta una sequenza di elementi. Amplia la classe Collection con altri metodi tra cui:

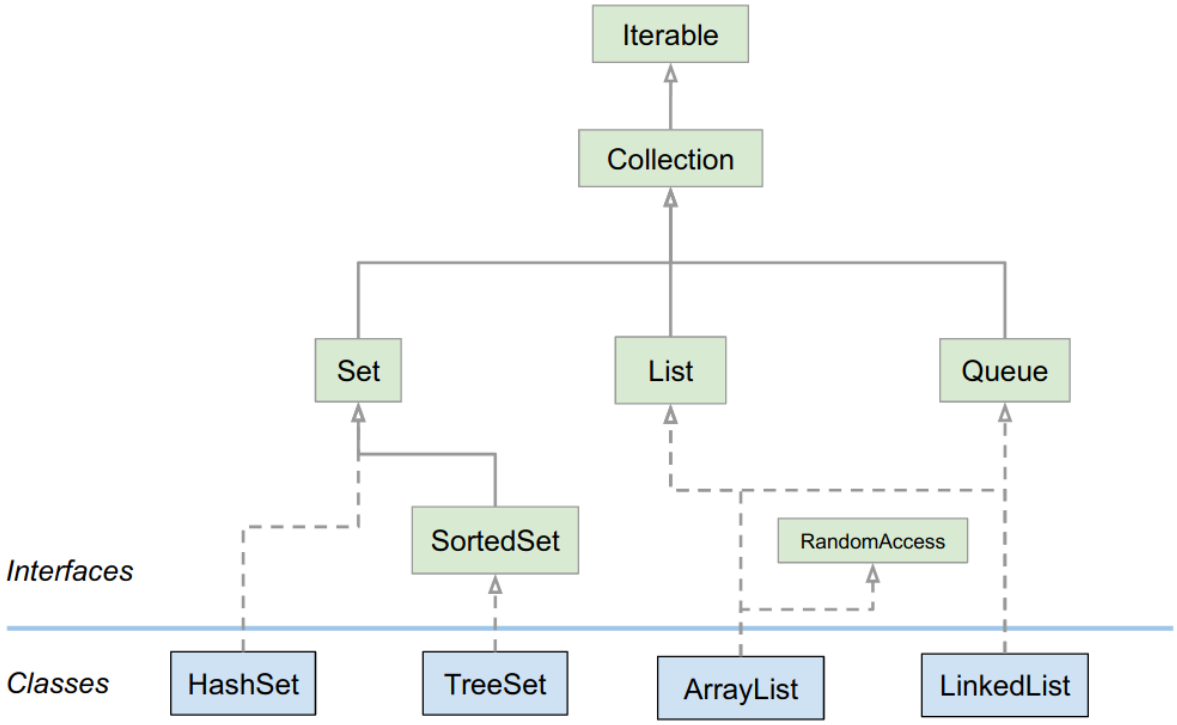
- `get(int i)`, che restituisce l'i-esimo elemento della sequenza (Solleva un'eccezione se l'indice è minore di zero o maggiore o uguale di `size()`);
- `set(int i, E elem)`, che sostituisce l'i-esimo elemento con `elem` (Solleva un eccezione se l'indice è scorretto)
  - NB: Non è possibile utilizzare set per allungare una lista

11.1.3 RandomAccess - Tag interface

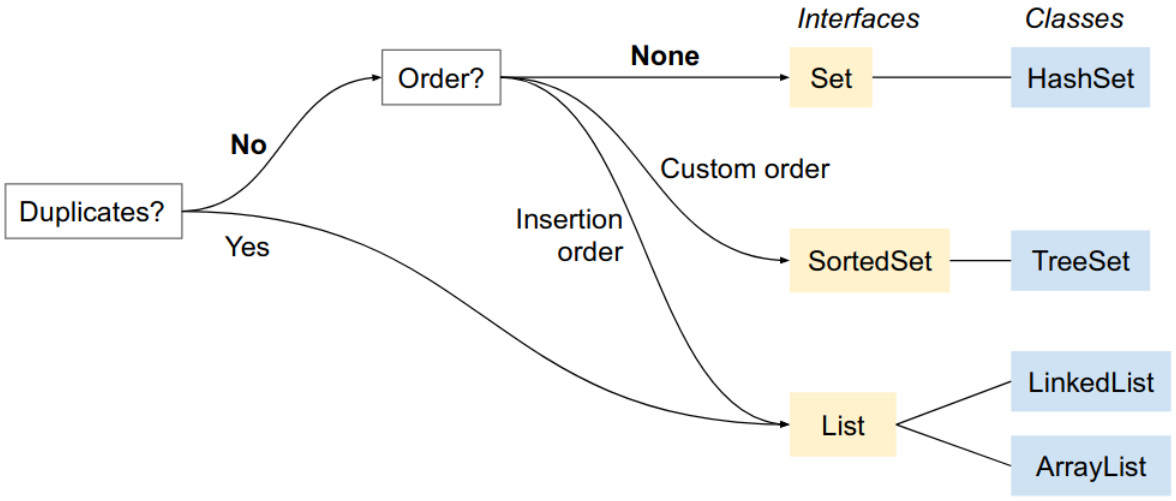
Per `tag interface` si intende un'interfaccia che ha il compito di marcare una classe per indicarne una proprietà astratta.

`RandomAccess` è un'interfaccia vuota, che serve a segnalare che la classe che la implementa offre l'accesso posizionale in maniera efficiente (ecco perché viene implementato da `ArrayList` e non da `LinkedList`).

11.2 Diagramma riassuntivo delle Collezioni



11.3 Scegliere una collezione



11.4 Esercizio - Inversione di una lista

```java
// NB: Dichiarare le loro variabili molto vicino al loro uso risulta molto comodo
// Si veda infatti l'oggetto temp
static <T> void reverse(List<T> l){
    if (l instanceof RandomAccess){
        for(int i = 0; i < l.size() / 2; i++){
            T temp = l.get(i);
            l.set(i, l.get(l.size() - i - 1));
            l.set(l.get(l.size() - i - 1), temp);
        }
    } else{
        List<T> temp = new Arraylist<>(l.size());
        // L'unico modo di modificare una collezione mentre la si scorre è solo tramite l'iteratore
        Iterator<T> i = l.iterator();
        while (i.hasNext()){
            temp.add(i.next());
            i.remove();
            for (int j = tem.size() -1; j >= 0; j --){
                l.add(temp.get(j));
            }
        }
    }
}
```

11.5 Complessità di `add` negli ArrayList

ArrayList è un'implementazione di List, realizzata internamente con un array di dimensione dinamica:

- Ovvero, quando l'array sottostante è pieno, esso viene riallocato con una dimensione maggiore, e i vecchi dati vengono copiati nel nuovo array

Questa operazione avviene in modo trasparente per l'utente.

Il metodo size restituisce il numero di elementi effettivamente presenti nella lista, non la dimensione dell'array sottostante

Il ridimensionamento avviene in modo che l'operazione di inserimento (add) abbia complessità ammortizzata costante (ricordiamo che si parla di media temporale).

$$T(n) = \text{costo di } n \text{ add}$$

Negli ArrayList:

- Capacità iniziale $\rightarrow$ 10;

- Fattore di crescita → `1.5` (La dimensione cresce del `50%`);

$$T(n) = 10 + 1 + 1 + \ldots + 1 + 15$$

Dove andiamo a rappresentare i primi `10 add` più una `grow` (15) seguiti da `5` add veloci e così via.

Notiamo che la somma di tutti gli `add` è `n` (Il problema sono i vari grow che siamo costretti a effettuare).

Avremo quindi che:

$$T(n) = n + \sum_{i=0}^{k} 10 \cdot (1.5)^i$$

`k` sarà il minimo valore che soddisfa la seguente equazione:

$$10 \cdot (1.5)^k > n$$

Che sviluppato sarà uguale a:

$$k > log_{1.5}(\tfrac{n}{10})$$

Sviluppato `k`:

$$n + \sum_{i=0}^{k} 10 \cdot (1.5)^i = n + 10 \sum_{i=0}^{log_{1.5}(\frac{n}{10})} (1.5)^i$$

Di conseguenza avremo che:

$$T(n) = n + 10\,\frac{1.5^{log_{1.5}(\frac{n}{10})+1}}{1.5 - 1}$$

Semplificando:

$$T(n) = n + 3n - 20 = \mathbb{O}(n)$$

11.6 Confronto sulla complessità dei metodi di LinkedList e ArrayList

Metodo	LinkedList	ArrayList
`add`	`O(1)`	`O(1)*`
`remove`	`O(n)`	`O(n)`
`contains`	`O(n)`	`O(n)`
`get`, `set`	`O(n)`	`O(1)`

`addFirst`
`addLast`
`removeFirst`
`removeLast`
	`O(1)`	`-`

Note:

- `(*)` → Complessità ammortizzata;
- `add` → Aggiunge in coda;
- `remove` → Deve trovare l'elemento prima di rimuoverlo.

11.7 Set

Rappresentano un *insieme* in senso matematico.

Non ammette duplicati (se infatti si tenta di aggiungere con add un elemento che è già presente, ovvero un oggetto che risulta uguale secondo *equals* ad uno già presente, la collezione non viene modificata e `add` restituisce `false`) e l'ordine in cui gli elementi vengono inseriti è irrilevante.

`Set` è un interfaccia che si divide in:

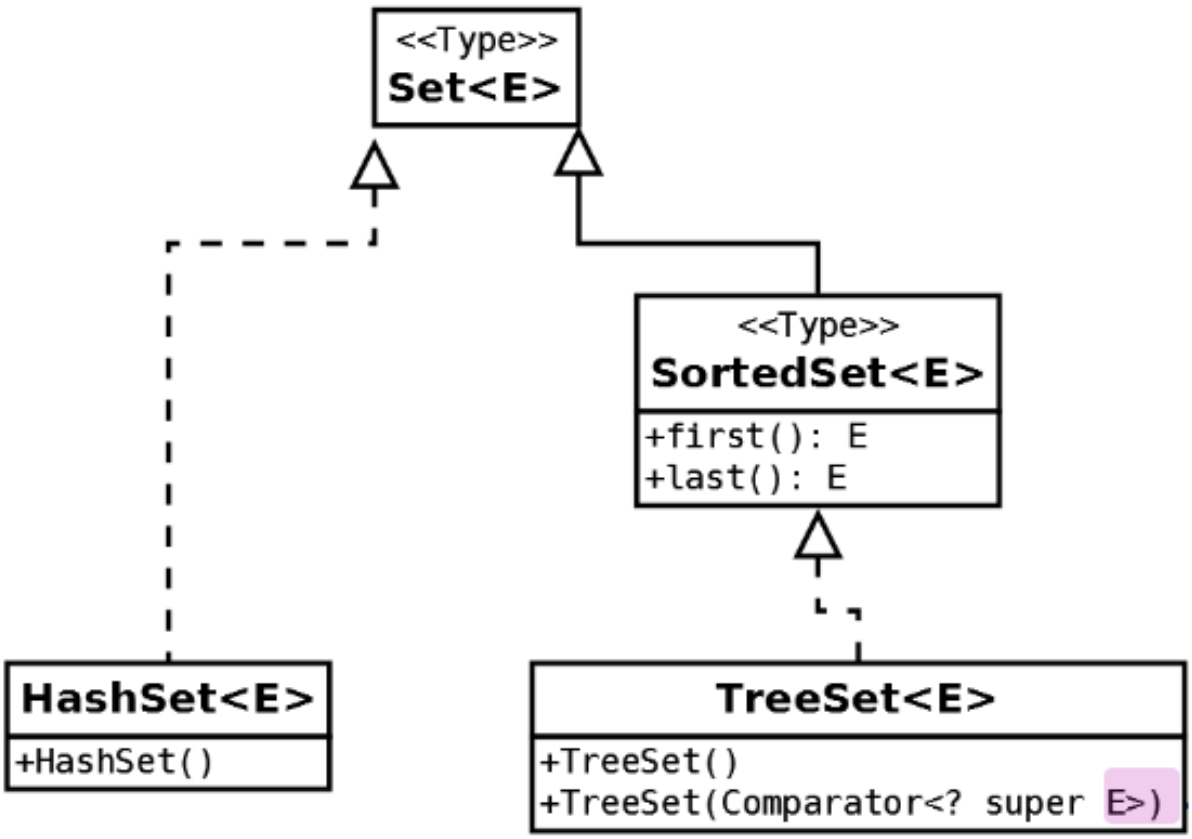
- `HashSet`;
- `SortedSet` → Garantisce che gli elementi saranno visitati in ordine (dal più piccolo al più grande)
  - A sua volta viene implementata da `TreeSet`, che è implementato interamente come albero di ricerca bilanciato.

Presenta 2 metodi:

- `T.first()` → Restituisce l'elemento minimo tra quelli presenti nella collezione;
- `T.last()` → Restituisce l'elemento massimo;

Entrambi i 2 metodi non modificano la collezione

Nel dettaglio l'albero di implementazioni di `Set`:



La differenza sostanziale tra Set e List è che il primo non raffina (non aggiunge metodi) il contratto di `Iterable`.

11.7.1 HashSet

È un Set realizzato internamente come `tabella hash`.

Utilizza il metodo hashCode della classe Object per selezionare il bucket in cui posizionare un elemento:

```
public int hashCode()
```

I principali metodi di `HashSet` hanno la seguente complessità media:

- `size` → `O(1)`;
- `isEmpty` → `O(1)`;
- `add` → `O(1)`;
- `contains` → `O(1)`;
- `remove` → `O(1)`;

Tuttavia, per le prestazioni reali pesa molto la bontà della funzione hash utilizzata.

`HashSet` utilizza il metodo equals per identificare gli elementi. Pertanto `€quals` e `hashCode` devono rispettare la seguente regola di coerenza:

$$\forall\ x \text{ e } y \text{ se } \texttt{x.equals(y)} \text{ è vero, allora } \texttt{x.hashCode == y.hashCode}$$

Una buona ridefinizione di `hashCode` deve rispettare le seguenti proprietà:

1. Coerenza con equals (necessario);
2. Coerenza temporale, cioè il valore dipende solo dallo stato dell'oggetto (necessario);
3. Uniformità, cioè il valore di ritorno è uniformemente distribuito sugli interi (desiderabile);

Perché HashSet funzioni in maniera efficiente, ed in particolare perché le operazioni principali abbiano complessità costante, è necessario che la classe componente (cioè, la classe degli elementi contenuti) disponga di un opportuno metodo hashCode.

11.7.2 Complessità computazionale delle operazioni di HashSet

size	`O(1)`
isEmpty	`O(1)`
add	`O(1)`
contains	`O(1)`
remove	`O(1)`

12 Lezione del 22-04

12.1 Program to Interface, not Implementation

Una buona regola di programmazione è utilizzare l'interfaccia più generica compatibile con le operazioni che devo compiere sull'oggetto che sto definendo:

```
1 // Modo corretto -> Mi permette di modificare in futuro il tipo concredo di app (in questo caso
      HashSet)
2 Set<Apparecchio> app = new HashSet<>();
3
4 // Definizione non corretta -> Limita la modifica futura
5 HashSet<Apparecchio> app = new HashSet<>();
```

12.2 TreeSet

Insieme, implementato internamente come **albero di ricerca bilanciato**.

Gli elementi devono essere dotati di una *relazione d'ordine*, in uno dei seguenti modi:

- Gli elementi sono dotati di ordinamento naturale → In questo caso è possibile utilizzare il costruttore senza argomenti;
- In caso contrario bisogna passare al costruttore di `TreeSet` un opportuno oggetto `Comparator` (`Comparator<? super E>`).

TreeSet utilizza un ordinamento, fornito tramite Comparable o da Comparator, per **smistare** e poi **ritrovare** gli elementi all'interno dell'albero. In particolare:

Se 2 oggetti sono equivalenti per l'ordinamento, saranno considerati uguali dal TreeSet

Allo stesso tempo, l'interfaccia `Set` prevede che si usi equals per identificare gli elementi.

Quindi, se l'ordinamento non è coerente con l'uguaglianza definita da equals, TreeSet può **violare il contratto di Set**.

Se vogliamo un TreeSet che rispetti il contratto di Set, dobbiamo fornire un ordinamento coerente con equals.

12.2.1 Costo operazioni principali di TreeSet

size	`O(1)`
isEmpty	`O(1)`
add	`O(log n)`
contains	`O(log n)`
remove	`O(log n)`
first	`O(1)`
last	`O(1)`

12.3 Problema della mutabilità degli elementi

Un altro problema che affligge sia TreeSet che HashSet riguarda la mutabilità degli elementi:

- Gli elementi vengono inseriti in queste strutture dati in base al valore dei loro campi
- Un TreeSet posiziona gli elementi sulla base di confronti con altri oggetti
- Un HashSet li posiziona sulla base del loro codice hash

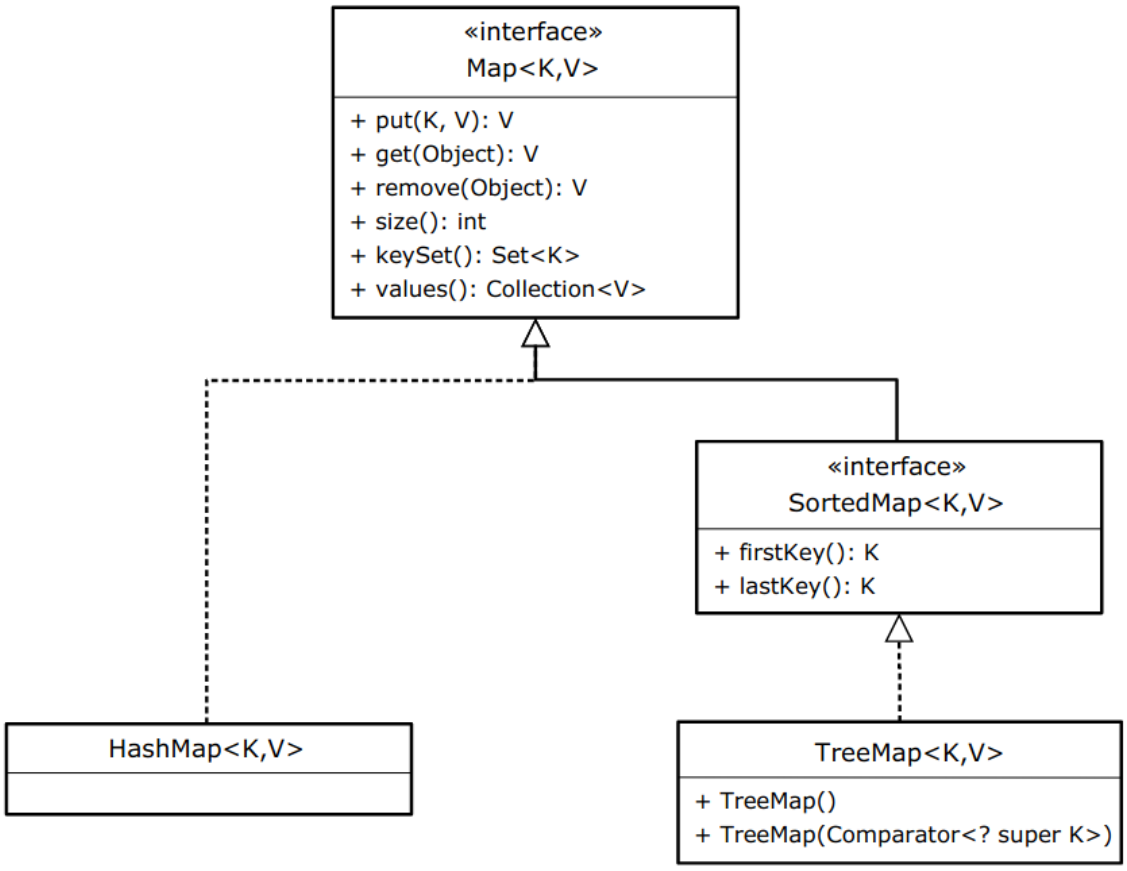
Se un elemento viene modificato dopo essere stato inserito in una di queste strutture dati, il posizionamento di quello specifico elemento non corrisponderà più al valore dei suoi campi → In questo caso, la struttura dati si comporterà in modo inatteso.

12.4 Java Collection Framework - Le mappe

Per mappa si intende un'insieme di coppie **chiave-valore**, senza chiavi duplicate.

In alcuni linguaggi, questo tipo di struttura dati prendere il nome di *dizionario* o *array associativo*.

Una mappa è analoga ad una funzione parziale da un insieme di chiavi ad uno di valori



Ricordiamo che nelle mappe sono consentiti duplicati (infatti più chiavi possono avere lo stesso valore).

Il modo consigliato per iterare su una mappa:

```
for(K key : map.KetSet()){
    V val = map.get(key);
    ...
}
```

13 Lezione del 29-04

13.1 Rompicapo con la ricorsione

```java
public class Recursion {
    public void test(){
        // In questo modo otteniamo l'eccezione `java.lang.StackOverflowError` (Esaurisce lo stack a disposizione)
        // Per ovviare a questo problema bisogna estendere la classe e sfruttare il concetto di `super`
        test();
        System.out.println("Recursion!!!");
    }
}

// Prima soluzione (Con classe anonima)
public class Test {
    new Recursion(){
        boolean firstTime = true
            @Override
            public void test(){
              if (firstTime){
                  firstTime = false;
                  super.test();
              }
            }
    }.test();

}


// Seconda soluzione (più elegante)

public class Test {
    new Recursion(){
        { super.test(); }
        public void test(){ }

    };
}
```

13.2 Le Enumeration

Tipi di dato con un numero finito e limitato di valori possibili

13.3 Le enumerazioni in Java

Fino alla versione 1.4, Java non offriva un simile supporto.

Per questo motivo una sua possibile implementazione doveva essere la seguente:

```
public class SuitClass {
    private SuitClass(String name){ this.name = name; }
    public final String name;

    public static final SuitClass HEARTS = new SuitClass"("Hearts);
    public static final SuitClass SPADES = new SuitClass"("Spades);
    public static final SuitClass CLUBS = new SuitClass"("Clubs);
    public static final SuitClass DIAMONDS = new SuitClass"("Diamonds);
}
```

Le regole da seguire sono le seguenti:

- Classe con soli costruttori privati;
- Ogni possibile valore del tipo enumerato corrisponderà ad una costante pubblica di classe (`public static final`);
- Ciascuna di queste costanti viene inizializzata usando uno dei costruttori privati.

A partire dalla versione 1.5 di Java, il linguaggio offre supporto nativo ai tipi enumerati, tramite il concetto di classe enumerata:

Una classe enumerata è un tipo particolare di classe, introdotta dalla parola chiave enum, che prevede un numero fisso e predeterminato di istanze

Riprendendo l'esempio precedente la classe `SuitClass` diventerà:

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

Una classe enumerata può consistere semplicemente di un elenco di valori possibili.

In particolare, i quattro valori dichiarati si utilizzano proprio come costanti pubbliche di classe, come, ad esempio in:

```
SuitEnum s = SuitEnum.HEARTS;
```

A differenza di SuitClass, l'ordine in cui i valori sono definiti in SuitEnum è **significativo**.

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

```
SuitEnum x = SuitEnum.DIAMONDS;
int i = x.ordinal();
String name = x.name();
SuitEnum[] allSuits = SuitEnum.values();
SuitEnum y = Enum.<SuitEnum>valueOf(SuitEnum.class, ""HEARTS)
```

Una classe enumerata può contenere:

- **Campi**;
- **Metodi**;
- **Costruttori**;

Le restrizioni di questo tipo di classe è sui costruttori:

- Devono avere visibilità `privata/default` (di pacchetto);
- Non è possibile invocarli esplicitamente con `new`, neanche all'interno della classe stessa.

Se una classe enumerata ha più costruttori, ciascun valore può essere costruito con un costruttore diverso.

Per le enumeration valgono le seguenti proprietà:

1. In una classe enumerata, la **prima riga** deve contenere l'elenco dei valori possibili;
2. Le classi enumerate estendono automaticamente la classe Parametrica **Enum** (che è possibile visualizzare al seguente link)
    - Le classi enumerate non possono estendere altre classi;
    - Ogni classe enumerata `E` estende `Enum<E>`
3. Le classi enumerate sono automaticamente `final`.

Ad ogni valore di una classe enumerata è associato un **numero intero** (chiamato **ordinale**) che rappresenta il suo posto nella sequenza dei valori, a partire da zero.

Per passare da **valore enumerato** a **ordinale** si usa il metodo messo a disposizione dalla classe Enum `public int ordinal()` (essendo pubblico questo metodo viene ereditato da tutte le classi enumerate).

Per l'operazione inversa, si usa il seguente metodo statico, che ogni classe enumerata `E` possiede automaticamente (non appartiene alla classe `Enum`):

```
public static E[] values()
```

Che restituisce un array contenente tutti i possibili valori di `E`.

Quindi per ottenere il valore di posto i-esimo, è sufficiente accedere all'elemento i-esimo dell'array restituito da `values`

È possibile anche passare da un valore enumerato alla stringa che contiene il suo nome, come definito nel codice sorgente, e viceversa.

Per passare da valore a stringa, si usa il metodo della classe Enum `public String name()`

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

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

Restituisce il valore enumerato della classe *enumType* che ha nome *name*.

Il parametro di tipo del metodo rappresenta la classe enumerata a cui lo si applica. *es*:

```
SuitEnum x = Enum.<SuitEnum>valueOf(SutiEnum.class, "HEARTS");
```

13.4 Specializzazione dei valori enumerati

È possibile specializzare il comportamento di un valore enumerato rispetto agli altri valori della stessa enumerazione.

In particolare, è possibile che un valore enumerato abbia una **versione particolare di un metodo** comune a tutta l'enumerazione. Ad *es* facendo riferimento alla classe precedente:

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

    public boolean isRed(){ return false; }
}
```

Per specializzare il comportamento di un valore, si inserisce il codice relativo subito dopo la dichiarazione di quel valore, racchiuso tra parentesi graffe.

Le versioni specializzate di `isRed` rappresentano un overriding del metodo presente in `SuitEnum`.

Le enumerazioni possono avere metodi astratti pur non essendo astratte esse stesse.

13.5 Collezioni per tipi enumerati

La JCF offre delle collezioni specificatamente progettate per le classi enumerate:

- `EnumSet`, versione specializzata di Set (ottimizzata per contenere elementi di una classe enumerata);
- `EnumMap`, versione specializzata di Map (ottimizzata per i casi in cui le chiavi appartengano ad una classe enumerata);

13.5.1 EnumSet

Intestazione completa:

```
1 // Il limite superiore del parametro di tipo impone che tale tipo estenda Enum di se stesso (
      requisito soddisfatto da tutte le classi enumerate)
2 public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E> implements Cloneable,
      Serializable
```

Internamente si presenta come **vettore di bit** → Se un EnumSet dovrà contenere elementi di una classe enumerata che prevede **n** valori possibili, esso conterrà internamente un vettore di *n valori booleani*.

L'i-esimo booleano sarà vero se l'i-esimo valore enumerato appartiene all'insieme (falso altrimenti).

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

È facile rendersi conto che questa tecnica implementativa non potrebbe funzionare su classi non enumerate, che non hanno un numero prefissato e limitato di valori possibili.

Questo implica che questa classe deve conoscere il numero di valori possibili che ospiterà. Per motivi di performance la libreria offre 2 diverse versioni (sottoclassi di EnumSet):

- JumboEnumSet, per enumerazioni con 64 elementi o meno;
- RegularEnumSet, per enumerazioni con più di 64 elementi;

Per questo motivo la classe `EnumSet` è astratta e non ha costruttori pubblici.

Per istanziarla, su utilizzano dei metodi statici, chiamati *metodi factory*

Uno di questi metodi factory è il seguente:

```
1 // Questo metodo crea un EnumSet vuoto, predisposto per contenere elementi della classe enumerata
      elemType
2 public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elemType)
```

Passare l'oggetto di tipo class corrispondente alla classe enumerata permette all'EnumSet di **conoscere il numero di valori possibili** che ospiterà.

13.5.2 EnumMap

Internamente, una EnumMap con valori di tipo *V* è semplicemente un **array di riferimenti di tipo V**.

L'ordinale delle chiavi funge da indice nell'array.

La sua intestazione completa è:

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

A differenza di `EnumSet`, presenta costruttori pubblici, tra cui:

```
// Crea una EnumMap vuota, predisposta per contenere chiavi della classe enumerata keyType
public EnumMap(Class<K> keyType)
```

14 Lezione del 06-05

14.1 La riflessione

È una caratteristica di Java che permette a i programmi di investigare a tempo di esecuzione sui tipi effetti degli oggetti manipolati.

Il cardine della riflessione è rappresentato dalla **classe Class**.

Ciascuno oggetto di classe Class rappresenta una delle classi del programma:

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

Pur non essendo classi, anche i tipi primitivi hanno un corrispondente oggetto di tipo Class.

14.2 Ottenere riferimenti agli oggetti di tipo Class

È possibile utilizzare 3 tecniche:

1. Metodo **getClass** della classe Object;
2. L'operatore **.class**;
3. Il metodo statico **forName** della classe Name;

14.2.1 Metodo getClass

Nella classe Object è presente il metodo:

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

che restituisce l'oggetto Class corrispondente al tipo **effettivo** di questo oggetto.

La classe Class ha un parametro di tipo; come un serpente che si morde la coda, il parametro di tipo di un oggetto Class indica il tipo che questo oggetto rappresenta. Ad *es.* se *x* è l'oggetto Class relativo alla classe `Employee`, il tipo di `x` è `Class<Employee>`.

14.3 Alcuni metodi della classe Class

```java
// Restituisce il nome di questa classe, completo di eventuali nomi di pacchetti (fully qualified)
public String getName();

// Crea e restituisce un nuovo oggetto di questa classe, invocando un costruttore senza argomenti, che questa classe deve possedere
public T newInstance();

// Restituisce l'oggetto Class corrispondente alla classe di nome `name`
// La stringa deve contenere anche l'indicizzazione degli eventuali pacchetti cui la classe appartiene.
public static Class<?> forName(String name);

// Restituisce l'oggetto Class corrispondente alla superclasse diretta di questa.
// Se questa è Object, oppure è un'interfaccia o un tipo base, il metodo restituisce null
public class<? super T> getSuperclass();

// Restituisce `true` se (e solo se) il tipo effettivo di `x` è sottotipo di questa classe
// È la versione riflessiva di instanceof
// È dinamico in entrambi i sensi (È la versione dinamica/generale di instanceof, che invece è dinamico solo su expr, ovvero solo sull'operando sinistro)
public boolean isInstance(Object x);
```

14.4 Tipo di ritorno di getClass

Il tipo restituito di getClass dovrebbe esprimere il seguente concetto

> Applicato ad un'espressione di tipo dichiarato `A`, questo metodo restituisce un oggetto di tipo `Class<? extends A>`

Non è possibile però esprimere in Java questa proprietà → Quindi, il type-checker tratta il metodo getClass in modo particolare,

simulando il tipo restituito che il linguaggio non è in grado di esprimere (fino a Java 5, il tipo di ritorno era `Class<? extends Object>`).

Precisamente il tipo di ritorno di `exp.getClass()` è l'erasure del tipo dichiarato dell'espressione `exp`.

14.5 L'operatore .class

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

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

Risulta quindi essere un *operatore unario postfisso*.

Ha carattere statico (il suo valore è noto al momento della compilazione).

14.6 Il metodo forName

Questa tecnica ha carattere dinamico, in quanto il valore restituito da forName non è noto al momento della compilazione.

14.7 Esempio

Di seguito presentiamo un metodo che accetta un array e un oggetto Class e riempi l'array di nuove istanze della classe corrispondente

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

Una sua possibile invocazione potrebbe essere la seguente:

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

Il metodo `newInstance` può lanciare diverse eccezioni verificate, nei casi in cui:

- La classe in questione non possegga un costruttore senza argomenti;
- Se tale costruttore non sia accessibile.

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

Sfortunatamente anche l'invocazione `<Object>fill(a, String.class)` è lecita.

14.8 Riflessione vs Generics

Consideriamo una classe per coppie di oggetti dello stesso tipo:

```
public class Pair<S>{
    private S first, second;
    public Pair(S a, S b){
        first = a;
        second = b;
    }
    public void setFirst(S a){ first = a; }
    public void getFirst(){ return first; }
    @Override
    public boolean equals(Object other){
        // In questo caso siamo costretti a usare la classe grezza
        if(!(other instanceof Pair)){
            return false;
        }
        Pair<?> p = (Pair) other;
        return first.equals(p.first) && second.equals(p.second);
    }
}
```

Adesso invece proviamo a simulare i generics con la riflessione:

```
public class Pair{
    private Object first, second;
    private final Class<?> type;
    public Pair(Class<?> c, Object a, Object b){
        // Controlliamo a run time quello che i generics controllano in fase di compilazione
        if (!c.isInstance(a) || !c.isInstance(b)) throw new IllegalArgumentException();
        type = c;
        first = a;
        second = b
    }
    public void setFirst(Object a){
        if(!type.isInstance(a)) throw new IllegalArgumentException();
        first = a;
    }
    public Object getFirst() { return first; }

    public boolean equals(Object other){
```

```
        if(!(other instanceof Pair)){
            return false;
        }
        Pair<?> p = (Pair) other;
    if (p.type != type) // Possiamo controllare il tipo effettivo di un'altra coppia
        return false;
        return first.equals(p.first) && second.equals(p.second);
    }
}
```

È importante notare che, una coppia di Manager risulta diversa da una coppia di Employee, anche se contengono gli stessi oggetti (2 Manager).

14.8.1 Combinazione dei Generics e Riflessione

```
public class Pair<S>{
    private S first, second;
    private final Class<?> type;

    public Pair(Class<?> c, S a, S b){
        type = c;
        first = a;
        second = b
    }

    public void setFirst(S a){ first = a; }
    public S getFirst(){ return first }
}
```

Per utilizzare questa classe:

```
// Corretta
Pair<String> p = new Pair<>(String.class, "uno", "due"); // "tre" Alza il volume nella testa /s
```

14.9 Ottenere informazioni su una classe

I metodi della classe Class permettono di ricavare numerose informazioni sulla classe in questione.

È possibile conoscere l'elenco di tutti i campi, metodi e costruttori apparenti alla classe.

A tale scopo, esistono le classi

Field, Method e Constructor

che rappresentano gli elementi omonimi di una classe.

Per ottenere tali informazioni è possibile fare affidamento ai seguenti metodi di Class:

```
// Restituisc tutti i campi pubblici di questa classe, anche ereditati
public Field[] getFields();
public Field[] getDeclaredFields();

// Analogo a `getFields`
public Method[] getMethods();
public Method[] getDeclaredMethods;

// Restituisce semplicemente i costruttori pubblici di questa classe
public Constructor[] getConstructors();
public Constructor[] getDeclaredConstructors;
```

14.9.1 Classe Field

Rappresenta un campo di una classe. Essa dispone di metodi per leggere e modificare il contenuto di un campo, conoscere il suo nome e il suo tipo. In particolare abbiamo:

`public String getName()`	Restituisce il nome di questo campo
`public Object get(Object x)` `throws IllegalAccessException`	Restituisce il valore di questo campo nell'oggetto `x` Se questo campo è un tipo base, il suo valore viene racchiuso nel corrispondente tipo riferimento Se questo campo è statico, il parametro `x` viene ignorato
`public void set(Object x, Object val)` `throws IllegalAccessException`	imposta a `val` il valore di questo campo nell'oggetto `x`. Se questo p un campo statico, il parametro `x` viene ignorato
`public Class<?> getType()`	Restituisce il tipo di questo campo

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

14.9.2 Sicurezza degli accessi

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

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

La possibilità di aggirare i controlli è soggetta al `Security Manager`, l'oggetto che è responsabile dei permessi per le operazioni a rischio.

Di default, la JVM si avvia senza un `Security Manager`, consentendo alle applicazioni di compiere qualsiasi operazione.

Invece, i browser che eseguono applet Java utilizzano dei `Security Manager` particolarmente restrittivi.

14.9.3 Classe Method

Rappresenta un metodo di una classe.

Dispone di metodi per conoscere il nome del metodo, il numero e tipo di parametri formali e il tipo di ritorno. È possibile invocare il metodo stesso.

In particolare abbiamo:

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

La sintassi `Object ... args` (varargs) indica che `invoke` accetta un numero variabile di argomenti.

14.10 Metodi variadici

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

Se `T` è un tipo di dati, con la scrittura

```
f(T ... x)
```

Si indica che `f` accetta un numero variabile di argomenti (anche 0), tutti di tipo T.

NB: I puntini sospensivi devono essere necessariamente 3.

Gli argomenti possono essere passati separatamente, come in `f(x1, x2, x3)`, oppure tramite un array, come in `f(new T[] {x1, x2, x3})`

All'intero del metodo `f`, si può accedere agli argomenti utilizzando `x` come un array di tipo `T`.

Ogni metodo può avere un solo argomento variadico, che deve essere l'ultimo della lista.

14.10.1 Riflessione in altri linguaggi

Il `C` non fornisce alcun supporto alla riflessione, mentre il `C++` ne fornisce un supporto parziale, chiamato `Run-Time Type Information` (RTTI).

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

In C, nessun oggetto (ad *es.* `struct`) conosce il proprio tipo, al contrario di `Java`, dove tutti gli oggetti conoscono il proprio tipo.

In `C++`, RTTI agisce solo su classi che hanno *almeno un metodo virtuale* (cioè di cui è possibile effettuare l'override):

- Gli oggetti di queste classi conoscono il proprio tipo, per permettere il binding dinamico;
- Gli oggetti delle altre classi non conoscono il proprio tipo.

14.10.2 Alcuni operatori del C++

Operatore typeid(exp)

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

Dynamic cast

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

15 Lezione del 09-05

15.1 Scegliere l'interfaccia di un metodo

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

- Nome del metodo;
- Lista dei parametri formali;
- Tipo di ritorno;

- Modificatori (visibilità, *static*, *final*, …);
- Un eventuale clausola *throws* (in Java).

Alcuni autori usano il termine *method header* per riferirsi all'interfaccia di un metodo.

In C++, si usa il termine prototipo per indicare invece il nome del metodo o funzione.

Per **firma** (signature) di un metodo si intende invece il nome del metodo e la lista dei parametri formali.

La firma contiene le uniche informazioni che sono rilevanti ai fini del binding dinamico.

Scegliere oculatamente l'interfaccia migliore per un metodo è particolarmente rilevante in alcuni contesti, tra cui:

- Metodi che manipolano **collezioni**
  - Queste presentano una ricca gerarchia di classi ed interfacce parametriche, rendendo complessa la scelta dell'interfaccia migliore per un metodo
- Metodi che appartengono a **librerie** destinate ad un ampio uso
  - Per un loro successo, è fondamentale che le interfacce dei metodi, e più in generale l'interfaccia pubblica della libreria, sia ben progettata, in modo da essere utile in quanti più contesti è possibile.

15.2 Scelta dei parametri formali

Dovrebbe **rispecchiare il più fedelmente possibile la pre-condizione** del metodo in questione.

Il tipo scelto dovrebbe:

- Accettare tutti i valori che soddisfano la pre-condizione (completezza);
- Rifiutare tutti gli altri valori (correttezza);

Questo è spesso impedito dai limiti del linguaggio di programmazione usato.

15.3 Correttezza VS Completezza

Quando non esiste una scelta corretta e completa, si deve scegliere tra 2 opzioni:

1. Scartare tutti i valori non validi ed anche alcuni valori validi (firma corretta ma non completa);
2. Accettare tutti i valori validi ed anche alcuni valori non validi (firma completa ma non corretta).

Di norma si preferisce la seconda scelta, in quanto i valori non validi che vengono accettati vengono scartati successivamente a run-time (lanciando una eccezione).

Pertanto in linea di massima assegneremo alla completezza una priorità **più alta** della correttezza.

In un contesto in cui sia prioritario la robustezza e quindi si voglia minimizzare il rischio di errori a run-time potrebbe essere preferibile la prima scelta.

15.4 Violare la completezza

Il criterio di completezza (o generalità) richiede che il metodo accetti tutti i valori che soddisfano la pre-condizione.

Violarla rende il metodo meno utile. in quanto non è applicabile a tutti i valori previsti dal contratto.

In altri termini, una firma non completa sta effettivamente `restringendo la pre-condizione`.

15.5 Violare la correttezza

Scegliere una firma che viola la correttezza comporta che il compilatore consentirà ai chiamanti di passare al metodo in questione dei valori non validi.

Violare questa condizione implica l'indebolimento della capacità del compilatore di verificare, tramite il type-checking, il rispetto delle pre-condizioni.

La parte di pre-condizione che non viene espressa nella firma dovrà essere verificata a run-time e potrà portare ad errori a tempo di esecuzione (lancio di eccezioni).

In altri termini, anche se la firma viola la correttezza, non si stia modificando (indebolendo) la pre-condizione.

15.6 Funzionalità

La firma di un metodo non dovrebbe mai accettare un tipo così generico da impedire di portare a termine il compito assegnato al metodo.

È possibile scegliere una firma che violi la correttezza, a patto di preservare la **funzionalità** del metodo stesso. Il tipo scelto deve contenere le informazioni (campi) e le funzionalità (metodi) necessarie a svolgere il compito previsto.

Nel valutare la funzionalità assumiamo di non voler ricorrere a conversioni forzate (cast).

15.7 Esprimere ulteriori garanzie

In alcuni casi, è possibile esprimere nel tipo dei parametri formali delle **garanzie offerte dalla post-condizione**, come il fatto che un dato parametro non venga modificato.

Un esempio di questo caso è nel `C++` che mette a disposizione il modificatore `const`, che serve proprio ad esprimere questo vincolo

Se, ad *es*, `Person` è una classe, un parametro di tipo `const Person&` è un riferimento ad un oggetto `Person`, che non può

essere utilizzato per modificare l'oggetto (In teoria il riferimento in questione può essere usato solo per invocare metodi che siano a loro volta dichiarati const).

In Java il tipo `jolly` può esprimere una proprietà simile nel caso delle collezioni.

Un parametro di tipo `Collection<? extends Employee>` sostanzialmente non può essere utilizzato per modificare la collezione. Più precisamente:

- Non è possibile invocare il metodo `add`, se non con argomento `null`;
- È possibile però invocare il metodo `remove`, perché il suo argomento è `Object`.

Analogamente un parametro di tipo `Collection<? extends Employee>` non può essere utilizzato per leggere il contenuto della collezione. Più precisamente:

- Si può accedere agli oggetti contenuti soltanto come se fossero degli `Object`

15.8 Criterio di semplicità

È preferibile un'interfaccia che sia più semplice da leggere e comprendere.

Nel caso dei metodi parametrici, questo criterio implica che, a parità delle altre caratteristiche, è preferibile un'interfaccia che utilizza **meno parametri di tipo**.

15.9 Scelta dei parametri formali

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

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

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

15.10 Scelta del tipo di ritorno

L'unica scelta da prendere riguarda la specificità:

- Un tipo di ritorno `più specifico` è più utile per il chiamante, perché esprime più informazioni sul valore restituito;
- Un tipo di ritorno `meno specifico`, nasconde l'implementazione interna del metodo (incapsulamento) e quindi favorisce l'evoluzione futura del Software.

Il progettista deve trovare un compromesso tra le 2.

15.10.1 Best pratice

Scegliere il tipo di ritorno in modo che conservi l'informazione di tipo presente nei parametri formali, ma nasconda i dettagli implementativi del metodo.

15.11 Tipo di ritorno e parametri formali

Si sceglie solitamente di dare più importanza al tipo di ritorno

16 Lezione del 13-05

16.1 Intersezione insiemistica - Esempio

Consideriamo il problema di scegliere l'interfaccia di un metodo che accetta 2 insiemi (Set) e restituisce l'intersezione dei 2.

È opportuno chiarire il contratto del metodo e in particolare la sua pre-condizione.

Se possibile dovremmo accettare come argomenti 2 insiemi di qualsiasi tipo.

In alcuni casi sappiamo a priori che gli insiemi non possano contenere elementi in comune:

- Non ha senso calcolarne l'intersezione, perché è sicuramente vuota;
- Potremmo quindi scegliere di scartare questo tipo di argomenti.

Tuttavia, si consideri il caso di un `Set<Cloneable>` e un `Set<Comparable<?>>`

- Anche se le interfacce Cloneable e Comparable non hanno alcun collegamento, niente impedisce che tali insiemi abbiano degli elementi in comune (oggetti che implementano entrambe le interfacce);
- Quindi, dovremmo accettare questo tipo di argomenti.

16.1.1 Firme dei metodi

```java
// Funzionale, corretta, ma non completa (Rifiuta insiemi di tipo diverso)
<T> Set<T> intersection(Set<T> a, Set<T> b);

// Funzionale, corretta, ma non completa (Rifiuta insiemi di tipo non correllato)
<T> Set<T> intersection(Set<T> a, Set<? extends T> b);

// Funzionale, corretta, completa (Espirme forti garanzia su entrambi i suoi argomenti)
// Il tipo di ritorno non ha alcuna relazione con quello dei 2 argomenti ed è poco utile al
    chiamante
Set<?> intersection(Set<?> a, Set<?> b);


// Funzionale, corretta, completa (Non esprime garanzie sugli argomenti)
// Il tipo di ritorno è più specifico (quindi migliore del caso precedente) in quanto conserva il
    tipo del primo argomento
<S,T> Set<S> intersection(Set<S> a, Set<T> b);

// Simile alla precedente, ma usa un parametro di tipo in meno e esprime ulteriori garanzie su b
<S> Set<S> intersection(Set<S> a, Set<?> b);
```

In base ai criteri presentati, l'interfaccia migliore risulta l'ultima in quanto:

- È funzionale, completa e corretta;
- Preserva nel tipo di ritorno una parte dell'informazione di tipo presente negli argomenti;
- Esprime la garanzia che il secondo argomento non venga ne scritto ne letto;
- Usa un solo parametro di tipo.

16.1.2 Esempio concreto

Per completezza esaminiamo il metodo di intersezione fornito dalla libreria `Google Guava` nella classe Sets:

```java
public static <E> Sets.SetView<E> intersetion(Set<E> a, Set<?> b)
```

NB: `Set.SetView` è un'interfaccia che estende `Set` e rappresenta una vista su un insieme.

La struttura è proprio quella dell'ultima firma da noi presentata.

L' API Java offre il seguente metodo nell'interfaccia `Collection<E>`:

```java
public boolean retainAll(Collection<?> other)
```

Il metodo modifica questa collezione, lasciando solo gli elementi presenti nella collezione `other`. Quindi effettua l'intersezione tra `this` e `other`.

In questo caso viene accettato un insieme qualunque tipo.

16.2 Approfondimenti

Per approfondire, è possibile consultare How to Design a Good API and Why it Matters, un seminario del `software enginerr` Joshua Bloch.

17 Lezione del 16-05

17.1 Union-find trees

Struttura dati (alberi) per la rappresentazione di *gruppi disgiunti*. Dato un insieme di elementi $e_1, \ldots, e_n$:

- Union operation → Dati 2 elementi, ne faccio la loro fusione;
- Find operation → Dato un elemento, otteniamo il rappresentante di un gruppo (Per controllare che 2 elementi siano nello stesso gruppo, controllo se il loro rappresentante è lo stesso)
    - Il rappresentante è la radice dell'albero che rappresenta quel gruppo.

17.1.1 Implementazione

Ciascun gruppo ha un puntatore al proprio padre (`parent-pointer tree`).

Nel caso dell'esempio del prof, ciascun nodo ha:

- Amount;
- Parent;
- Size.

17.1.2 Path compression

Quando si naviga da un nodo alla root, si trasforma ciascuno nodo insieme al path in un figlio diretto della root.

17.1.3 Link-by-size policy

Quando vengono fusi 2 alberi, viene linkato l'albero più piccolo alla root del più grande. Questo metodo garantisce che l'altezza finale dell'albero sarà al più logaritmica nel numero di nodi.

17.1.4 Teorema di Tarjan

Qualsiasi sequenza di *m* operazioni union o find su *n* elementi richiede al massimo tempo $\Omega(m\ \alpha(n))$, dove $\alpha()$ è la funzione inversa di **Ackermann**.

- $\alpha(n)$ è al più 4 per tutti gli n a 10^{80};
- Il costo di ogni singola operazione è sostanzialmente `costante`.

17.2 Efficienza di spazio

È importante considerare questo punto. Ci si potrebbe trovare nella situazione in cui si abbia poco spazio a disposizione o avere dati in grande quantità.

17.2.1 Object layout

Gli oggetti in Java contengono un *object header* che contiene informazioni ausiliare.

Ogni oggetto deve conoscere il suo tipo effettivo. L'header contiene un puntatore al tipo effettivo

17.2.2 Overhead dovuto al multithreading

Java assegna un monitor a ogni oggetto

Simile ad un mutex, e accessibile dalla keyword `synchronized`

17.2.3 Overhead dovuti al garbage collection

Tecniche di mark-and-sweep (gli oggetti non hanno overhead, ma la garbage collection parte delle radici) e generazionali (gli oggetti sono raggruppati in gruppi chiamati generazione, basata sul tempo di creazione).

La seconda tecnica richiede un piccolo overhead per aggiungere l'informazione della data di creazione.

17.2.4 Overhead dovuto all'allineamento e padding

In molte architetture, gli accessi di memoria sono molto più efficienti se sono `word-aligned`.

In una architettura a 64 bit, un indirizzo è `word-aligned` se multiplo di 8.

Se necessario viene aggiunto spazio vuoto (padding).

18 Lezione del 20-05

18.1 I principi di Lambda-calculus

Un linguaggio di funzioni che partendo da qualche input porta a qualche output.

Di tipo stateless → puro, assenza di risultati inaspettati.

Assenza di variabili, assegnamenti e cicli.

18.2 Linguaggi funzionali

- Lisp;
- Haskell;
- ML

18.3 Parallelismo funzionale

Composizioni invece di comunicazione:

```
f(g(a), h(b), g(c))
```

Le funzioni stateless `g` ed `h` possono essere valutate in qualsiasi ordine (anche parallelo).

La comunicazione avviene solo attraverso valori di ritorno.

Assenza di `race conditions` → Nessuna necessità di sincronizzazione.

18.4 Interfacce vs Classi astratte

Le interfacce sono ancora stateless (non presentano attributi) → offrono un comportamento, non uno stato.

Le classi astratte offrono comportamento e stato.

18.5 Evoluzione del linguaggi e compatibilità

È possibile aggiungere metodi statici e default ad una interfaccia senza intaccare le classi che lo implementano.

Molte interfacce standard sono state in questo modo arricchite.

18.6 Interfacce funzionali

- Qualsiasi interfaccia con un singolo metodo astratto;
- Permessi metodi statici e default.

18.7 Interfacce puramente funzionali

Una interfaccia puramente funzionale che ha lo scopo di essere implementare da classi `stateless`.

Rispettano il paradigma funzionale. Giocano un ruolo fondamentale in congiunzione con gli *streams*.

18.8 L'annotazione FunctionalInterface

Creata per le interfacce puramente funzionali → Il compilatore controlla la proprietà di *singolo metodo astratto*.

18.9 Lambda espressioni

Alternativa alle classi anonime:

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

18.9.1 Sintassi

```
parameters -> body
```

18.10 Cattura dei valori

Le lambda espressioni possono accedere a:

- Campi statici di ogni classe (trivial);
- Variabili locali racchiuse in metodo (necessità di `capture`) purché siano `effectively final`;
- Stato degli attributi racchiusi in un oggetto (necessità di `capture`);

18.10.1 Implementazione della cattura di variabili locali

1. Memorizzano la copia della variabile;
2. Catturano la variabile;
3. Ogni valutazione a runtime può generare o meno un nuovo oggetto.

18.10.2 Implementazione della cattura di campi

1. Memorizzano un riferimento all'oggetto che li racchiude;
2. Catturano l'istanza corrente (`instance-capturing lambda expression`);
3. Simile alla cattura della variabile locale `this`;
4. Ogni valutazione a runtime genere un nuovo oggetto.

18.11 Variabili effectively final

Un tipo di variabile utilizzata come se fosse `final`. Non è riassegnabile.

18.12 Lambda espressioni vs Classi Anonime

- Le lambda espressioni sono più succinte e non creano file di classe aggiuntivi;
- Non tutte le occorrenze di una lambda genere un nuovo oggetto;
- Le classi anonime permettono molteplici metodi;
- Le classi anonime possono avere uno stato.

19 Lezione del 23-05

19.1 Introduzione al multi-threading

I **thread** (detti anche processi leggeri), sono flussi di esecuzione all'interno di un processo in corso.

Un processo, può essere suddiviso in vari **thread**, ciascuno dei quali rappresenta un flusso di esecuzione indipendente dagli altri.

I thread appartenenti allo stesso processo condividono quasi tutte le risorse, come la memoria e i file aperti, tranne:

- Program counter;
- Stack.

Entrambe risorse che consentono ad un thread di avere un flusso di esecuzione indipendente.

Java stato il primo tra i linguaggi di programmazione utilizzati ad offrire un supporto nativo ai thread.

In altri linguaggi, come il C++, è necessario introdurre librerie esterne, spesso fornite dal SO.

Siccome la JVM funge anche da sistema operativo per i programmi Java, essa offre in maniera nativo il supporto ai thread.

19.2 Oggetti Thread e thread di esecuzione

In Java ad ogni thread di esecuzione è associato un oggetto thread.

Il viceversa non è sempre vero → Un oggetto thread può non avere un corrispondente thread di esecuzione (perché questo non è ancora partito o è già terminato).

19.3 Applicazioni e thread

Un'applicazione Java termina quanto tutti i suoi thread sono terminati.

Ogni applicazione Java parte con almeno thread, detto `main thread`, che esegue il metodo `main` della classe di partenza.

Anche al thread principale, è associato (in maniera automatica) un `oggetto thread`.

19.4 Creazione di un Thread

Un primo modo di creare un Thread è il seguente:

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

NB: All'occorrenza il procedimento può essere semplificato utilizzando una classe X anonima.

19.4.1 Esempio di creazione di un thread di esecuzione

```
1  public class MyThread extends Thread {
2      @Override
3      public void run(){
4          for (int i = 0; i < 10; i++){
5              System.out.println(i);
6              try{
7                  // Mette in attesa il thread corrente, per un dato numero di millisecondi
8                  Thread.sleep(1000); // Metodo bloccante
9              } catch ( InterrupetedException e){
10                 return; // Chiudiamo in questo modo il thread di esecuzione
11             }
12         }
13     }
14
15     // A questo punto, istanziamo la classe ~MyThread~ e facciamo partire il corrispondente thread di
           esecuzione
16
17     MyThread t = new MyThread();
18     t.start();
```

Non abbiamo dotato la classe MyThread di un costruttore in quanto `Thread` ha un costruttore senza argomenti.

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

Il metodo `start` non è bloccante → Il nuovo thread di esecuzione svolge le sue operazioni in parallelo al resto del programma.

Non è consentito invocare il metodo **`start`** più di una volta sullo stesso oggetto thread, anche se la prima esecuzione del thread è terminata.

Il nuovo thread di esecuzione esegue automaticamente il metodo **`run`** dell'oggetto thread corrispondente → Il metodo run è detto anche l' *entry point* del thread.

19.5 Altri metodi di Thread

```
// Restituisce l'oggetto thread corrispondente al thread di esecuzione che l'ha invocato
// È possibile con questo metodo anche ottenere un riferimento all'oggetto thread
// corrispondente al thread principale
public static Thread currentThread();

/*
  Interagisce con 2 thread (Sia oggetti, che thread di esecuzione)
  Il primo lo invoca sul secondo (Cioè il thread corrispondente all'oggetto puntato da this)
  Questo metodo mette in attesa il thread 1 fino alla terminazione del secondo
  Se il secondo non è partito o è già terminato, il metodo ritorna immediatamente
*/
public final void join() throws InterruptedException; // Metodo bloccante
```

Il metodo **`join`** è analogo alla system call **`waitpid`** dei sistemi Unix, e della funzione **`pthread_join`** dello standard POSIX thread.

19.6 Interruzione di un Thread

Risulta spesso utile interrompere le operazioni di un thread.

A tale scopo ogni thread è dotato di un flag booleano detto *stato di interruzione*, (di default falso).

I metodi bloccanti **`sleep`** e **`join`** vengono interrotti non appena lo stato di interruzione diventa vero.

Per modificare il valore dello stato di interruzione è presente il seguente metodo:

public void interrupt()

Nonostante il suo nome, questo metodo non ha effetto diretto su un thread di esecuzione.

In particolare, se tale thread non sta eseguendo un'operazione bloccante, la chiamata ad un interrupt non ha nessun effetto immediato.

La successiva chiamata bloccante troverà modificato lo stato di interruzione e pertanto uscirà immediatamente lanciando l'apposita eccezione.

19.7 Conoscere lo stato di interruzione di un thread

È possibile conoscere lo stato di interruzione di un thread attraverso il metodo

public boolean isInterrupted()

che restituisce l'attuale stato di interruzione di questo thread, senza modificarlo.

19.8 La disciplina delle interruzioni

Un'applicazione dovrebbe sempre essere in grado di terminare tutti i suoi thread su richiesta → In un ambiente interattivo, l'utente potrebbe richiedere la chiusura dell'applicazione in qualsiasi momento.

Per ottenere questo risultato, tutti i thread dovrebbero rispettare la seguente disciplina relativamente alle interruzione:

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

NB:

- Queste regole valgono soprattutto per quei thread che hanno potenzialmente illimitata, come quelli basati su un ciclo infinito
- In questo caso è buona pratica di sostituire a `while(true)` (e simili)
  - Un controllo del tipo `while(!Thread.currentThread.isInterrupted())`;
- È anche possibile segnalare in altro modo un'interruzione al thread in altro modo, utilizzando ad esempio lo stato di un oggetto condiviso.

19.9 Interfaccia Runnable

Utilizzare come superclasse Thread pone un grosso limite sull'estensione della nostra classe (spezziamo infatti la catena di estensione).

Il alternativa, è possibile creare un thread di esecuzione tramite una nostra classe che implementi l'interfaccia `Runnable`:

```
1 public interface Runnable{
2     public void run();
3 }
```

Questa interfaccia contiene solo un metodo `run` (Analogo a quello presente in `Thread`)

Questo metodo sarà l' *entry point* per il nuovo thread di esecuzione,

Per creare il thread, utilizziamo il costruttore della classe Thread che accetta come argomento un oggetto di una nostra classe che implementa Runnable:

public Thread(Runnable r)

Analogamente ai thread è necessario invocare il metodo `start`, per far partire il nuovo thread.

19.9.1 Thread creati con Runnable

È possibile utilizzare lo stesso oggetto Runnable per creare più thread, che eseguiranno lo stesso metodo `run`.

È importante ricordare che non ci si trova più a estendere la classe Thread → Pertanto non sarà possibile invocare direttamente `isInterrupted` e `sleep`, ma sarà necessario invocare:

- `Thread.currentThread().isInterrupted()`;
- `Thread.sleep(1000)`;

19.10 Tabella riassuntiva

Riassumiamo tutti i metodi della classe Thread (pubblici):

- `Thread()` → Costruttore senza argomenti;
- `Thread(Runnable r)` → Costruttore che accetta un Runnable;
- `void start()` → Crea e avvia il corrispondente thread di esecuzione;
- `void run()` → Entry point del thread;
- `void join() throws InterrupetedException` → Attende la terminazione di questo thread;
- `void interrupt()` → Imposta a true lo stato di interruzione di questo thread;
- `boolean isInterrupted()` → Restituisce lo stato di interruzione di questo thread;
- `static Thread currentThread()` → Restituisce l'oggetto thread del thread di esecuzione corrente;
- `static void sleep(long m) throws InterrupetedException` → Attende *m* millisecondi.

19.11 Comunicazione tra thread

Piuttosto che evolvere in maniere del tutto indipendente tra loro, è spesso utile che 2 thread possano comunicare.

Il modo più semplice è utilizzare **oggetti condivisi** → Si tratta semplicemente di oggetti ai quali entrambi i thread posseggono un riferimento.

Per far ciò si può fare uso delle collezioni fornite dalla libreria standard di Java (ricordiamo che tipi primitivi e Tipi wrapper non possono essere utilizzati, in quanto i primi vengono passati per valore e non per riferimento, mentre i secondi sono tipi immutabili), o attraverso gli oggetti di `concurrent` `atomic` messa a disposizione sempre dalla libreria standard

20 Lezione del 27-05

20.1 Sincronizzazione tra thread

Se due thread tentano di modificare contemporaneamente lo stesso oggetto, l'interleaving arbitrario stabilito dallo scheduler può far si che l'operazione lasci l'oggetto in uno stato incoerente:

- È necessario garantire che solo un thread alla volta possa modificare tale oggetto;
- Questa proprietà prende il nome di mutua esclusione;
- La soluzione classica al problema prevede l'uso di mutex;

20.1.1 Mutex

Semaforo binario che supporta le operazioni base di *lock* e *unlock*

Java integra i mutex nel linguaggio stesso:

- Ad ogni oggetto, indipendentemente dal tipo, è associato un mutex (detto *monitor*) e una corrispondente lista d'attesa;
- La keyword *synchronized* permette di utilizzare implicitamente tali mutex (questo modificatore è applicabile anche ad un metodo, o ad un blocco di codice, ma non ad un `campo/variabile`);

20.2 Metodi sincronizzati

```
public synchronized int f(int n) { ... }
```

Supponiamo che il metodo venga invocato con `x.f(3)`

In questo esempio, l'effetto del modificatore synchronized è il seguente:

- Prima di entrare nel metodo `f`, il thread corrente tenta di acquisire il mutex di *x*
  - Informalmente è come se il thread chiamasse `x.mutex.lock()`;
- Se il mutex è già impegnato, il thread viene messo in attesa che si liberi;
- Quando esce dal metodo `f`, il thread rilascia il mutex di *x*
  - Informalmente è come se il thread chiamasse `x.mutex.unlock()`;

In altre parole, quando un thread invoca un metodo sincronizzato *f* di un dato oggetto, altri thread che invochino qualunque metodo sincronizzato dello stesso oggetto devono aspettare che il primo thread esca dalla chiamata a *f* → Questo garantisce che solo un thread alla volta possa eseguire i metodi sincronizzati di ciascun oggetto.

Se un metodo **statico** di una classe *A* è sincronizzato, il thread che lo invoca acquisirà il mutex dell'oggetto **Class** corrispondente alla classe *A*.

20.2.1 synchronized e overriding

In caso di overriding, un metodo che era sincronizzato può diventare non sincronizzato (e viceversa). Quindi:

- Un'interfaccia non può forzare le sue implementazioni ad avere metodi sincronizzati;
    - Non è possibile quindi applicare ai metodi astratti di un'interfaccia la keyword synchronized.

20.2.2 Blocchi sincronizzati

La parola chiave synchronized può anche introdurre un blocco di codice:

- In questo caso, parleremo di blocco (di codice) sincronizzato;
- Usato in questo modo, synchronized richiede come argomento l'oggetto del quale vogliamo acquisire il mutex;

È importante notare che i mutex acquisiti dai blocchi sincronizzati sono gli stessi che sono utilizzati anche dai metodi sincronizzati.

Pertanto, se un thread sta eseguendo un blocco che è sincronizzato sull'oggetto *x*, gli altri thread devono aspettare per eseguire eventuali metodi sincronizzati di *x*

20.3 Osservazioni

I mutex impliciti di Java sono rientranti (reentrant) → Ciò significa che un thread può acquistare lo stesso mutex più volte.

Questo accade comunemente, ogni qual volta un metodo sincronizzato ne chiama un altro (anche esso sincronizzato).

Se i mutex non fossero rientranti, un metodo sincronizzato che ne invoca un altro sullo stesso oggetto andrebbe immediatamente in deadlock.

Internamente, un mutex rientrante ricorda quante volte è stato acquistato dallo stesso thread:

- Il mutex contiene un contatore, che viene incrementato ad ogni acquisizione (lock) e decrementato ad ogni rilascio (unlock);
- Il mutex risulta libero quando il contatore vale 0;

Per certi versi, un mutex rientrante è simile ad un semaforo (*counting semaphore*):

- Un semaforo può essere `incrementato/decrementato` da più thread, mentre un thread non può né acquisire né rilasciare un mutex che in quel momento risulti acquisito (una o più volte) da un altro thread.

20.4 Classi thread-safe

Da `Java Concurrency in Pratice`:

> A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code

Una classe thread-safe mantiene il proprio contratto anche se utilizzata da diversi thread contemporaneamente, senza sincronizzazione da parte del chiamante.

Un metodo sarà thread-safe se potrà essere invocato contemporaneamente da diversi thread, mantenendo sempre rispettato il suo contratto e l'invariante di classe

20.4.1 Esempio di thread-safety

```
// Non thread-safe: può violare il contratto di incrementSalary
public void incrementSalary(int delta){
    if (delta >=0){
        salary += delta;
    }
}

// Non thread-safe: può violare l'invariante e il contratto di incrementSalary
public void incrementSalary(int delta){
    if (salary + delta >=0){
        salary += delta;
    }
}

// Thread safe
public synchronized void incrementSalary(int delta){
    if (salary + delta >=0){
        salary += delta;
    }
}
```

20.5 Collezioni standard e thread safety

La maggior parte delle collezioni standard non è thread safe per motivi di efficienza. Non sono thread safe:

- LinkedList;
- ArrayList;
- `HashSet/Map`;
- `TreeSet/Map`.

Se più thread condividono una di queste collezioni, e almeno uno di questi modifica la collezione (è uno *scrittore*), tutti i thread devono accedere alla collezione in mutua esclusione → Acquisendo ad esempio il monitor della collezione.

20.6 Le condition variable

Le variabili di condizione sono un classico meccanismo di sincronizzazione, che permette ad un thread di attendere una condizione arbitraria, che altri thread renderanno vera.

Supponiamo che 2 thread condividano una variabile intera e che il primi thread debba aspettare che il secondo ne modifichi il valore per poter andare.

Una soluzione ingenua consiste nell'utilizzare una ulteriore variabile condivisa, di tipo booleana e un ciclo del tipo:

```
1 while (!modified) {/* ciclo vuoto */}
```

Questa soluzione prende il nome di *attesa attiva*, perché durante l'attesa il thread occupa inutilmente la CPU, controllando costantemente la condizione di uscita dal ciclo.

20.6.1 Attesa passiva

La soluzione corretta, consiste nel realizzare una *attesa passiva*, utilizzando una condition variable.

È un meccanismo di sincronizzazione che, unito ad un mutex, permette di attendere il verificarsi di una condizione tramite attesa passiva e senza rischi di *race condition*.

20.7 Le condition variable in Java

Come i mutex, anche le condition variable sono realizzate in modo implicito (senza utilizzare esplicitamente oggetti di tipo *condition variable*).

In particolare la loro funzionalità viene offerta dai seguenti metodi della classe Object:

```
1 // Intuitivamente, il metodo `wait`, chiamato su un oggetto `x`, mette il thread corrente in attesa
      che qualche altro thread chiami notify o notifyAll sull'oggetto `x`
```

```
// Quindi, l'oggetto `x` fa da tramite per permettere al secondo thread di comunicare al primo che
    può andare avanti nelle sue operazioni

public void wait() throws InterrupetedException; // operazione bloccante

public void notify();

public void notifyAll();
```

Come tutti i metodi bloccanti, `wait` è sensibile allo stato di interruzione del thread, e in caso di interruzione solleva l'eccezione verificata `InterruptedException`

La differenza tra `notify` e `notifyAll` è che il primo risveglia uno solo dei thread che sono potenzialmente in attesa della condizione, mentre notifyAll li sveglia tutti.

20.7.1 Funzionamento interno di wait

1. Se il thread corrente non possiede il monitor di x, lancia una eccezione;
2. Se lo stato di interruzione del thread corrente è vero, lancia un'eccezione;
3. In un'unica operazione atomica:
    - Mette il thread corrente nella lista di attesa di `x`;
    - Rilascia il monitor di `x`;
    - Sospende l'esecuzione del thread.

Se il thread viene risvegliato da `notify()` o `notifyAll()` o da un risveglio spurio:

- Riacquisisce il mutex di `x`;
- Restituisce il controllo al chiamante;

Se invece il thread viene interrotto:

- Riacquisisce il mutex di `x`;
- Lancia `InterrupetedException`.

20.7.2 Osservazioni e funzionamento sulle funzioni di notifica

1. Se il thread corrente non possiede il monitor di `x`, lancia una eccezione;
2. Se si tratta di notify
    - Preleva un thread dalla coda di attesa di `x` e lo rende nuovamente eseguibile;

3. Se si tratta di notifyAll
    - Preleva tutti i thread dalla coda di attesa di `x` e li rende nuovamente eseguibili.

20.8 Applicazione delle condition variable

Una soluzione ricorrente nella programmazione concorrente è nel paradigma **produttore-consumatore**.

Si tratta di 2 o più thread, divisi in 2 categorie:

- I produttori, sono fondi di informazioni destinate ai consumatori;
- I consumatori devono elaborare le informazioni fornite dai produttori, non appena queste si rendono disponibili.

Non è possibile prevedere quanto tempo impiega un produttore a produrre un'informazione, né quanto impiega un consumatore ad elaborarla.

Si pone il problema di **sincronizzare le operazioni** tra le 2 categorie.

La soluzione classica prevede uno o più *buffer*, che contengono le informazioni prodotte e non ancora consumate.

Le condition variable permettono di realizzare queste attese in modo passivo e senza rischio di race condition.

20.8.1 Esempio

Supponiamo che il riferimento `buf` punti ad una struttura dati con metodi:

- `put` → aggiunge un elemento;
- `take` → rimuove un elemento;

I 2 thread seguenti usano il buffer non solo per comunicare, ma anche per sincronizzarsi

Produttore e Consumatore

```
synchronized (buf){
    // attende che il buffer non sia pieno
    while (buf.isFull()){
        try{
            buf.wait();
        } catch (InterrupetedException e){
            return;
        }
    }
    buf.put(some_value);
    // notifica i consumatori
    buf.notifyAll();
}
```

```
synchronized (buf){
    // attende che il buffer non sia pieno
    while (buf.isEmpty()){
        try{
            buf.wait();
        } catch (InterrupetedException e){
            return;
        }
    }
    some_value = buf.take
    // notifica i produttori
    buf.notifyAll();
}
```

20.8.2 Osservazioni

Viene da chiedersi perché sia il produttore che il consumatore basano la loro attesa su un ciclo, invece di una semplice condizione.

Utilizzando la seconda opzione si possono presentare 3 problemi:

1. Se il produttore utilizza `notifyAll` → Vengono svegliati 2 consumatori, ma esiste un solo valore nel buffer;
2. Se il produttore utilizza `notify` → Viene svegliato un solo consumatore, ma un altro lo anticipa;
3. In tutti i casi → Un risveglio spurio da wait può portare a leggere da un buffer vuoto

21 Lezione del 30-05

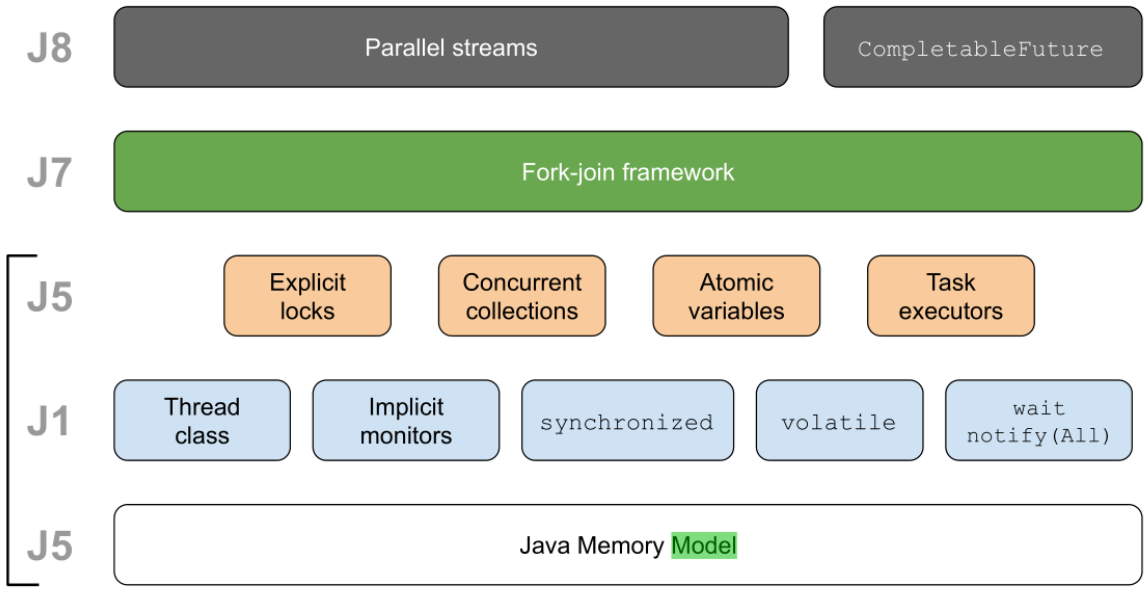
21.1 Supporto ai thread in C++

```
// Da compilare con `g++ -o nome_eseguibile nome_sorgente.cpp -pthread`
#include <thread>
#include <iostream>

using namespace std;

void foo(int n, char* args[]){
  for (int i = 0; i<n ; i++){
    cout << i << "\t" << args[i] << endl;
  }
}

int main(int argc, char *argv[]){
  thread t1(foo, argc, argv);
  thread t2(foo, argc, argv);
```

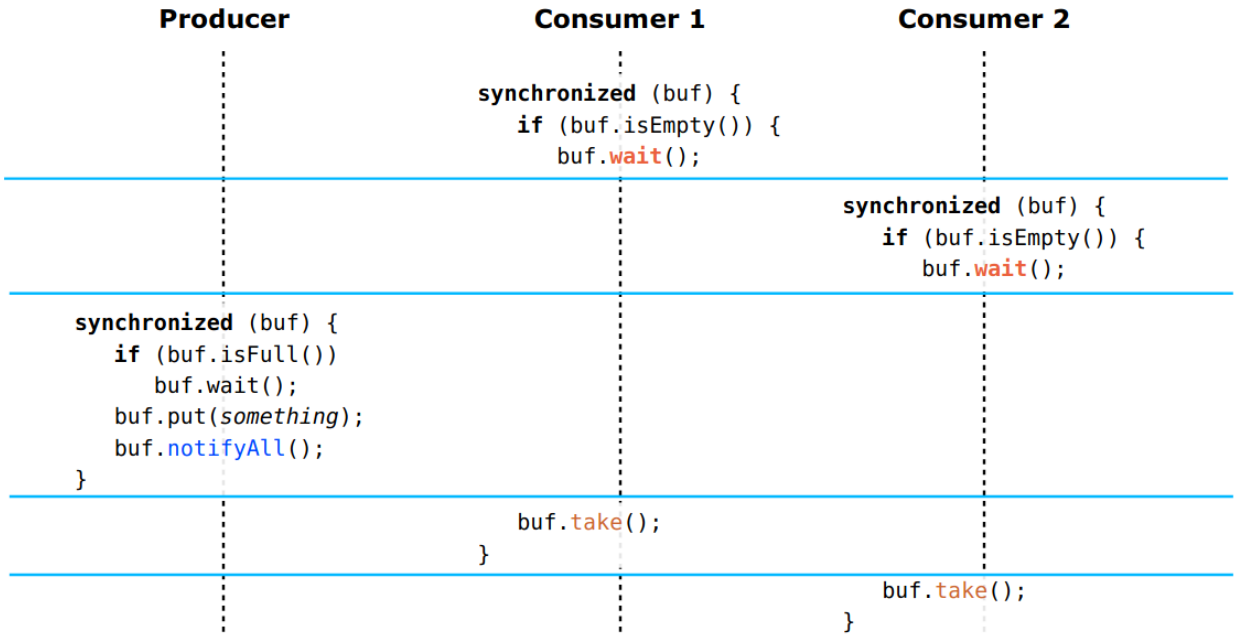
```
  thread t3(foo, argc, argv);
  t1.join();
  t2.join();
  t3.join();
}
```

Differenza sostanziale è la mancanza del metodo start sui singoli `thread`.

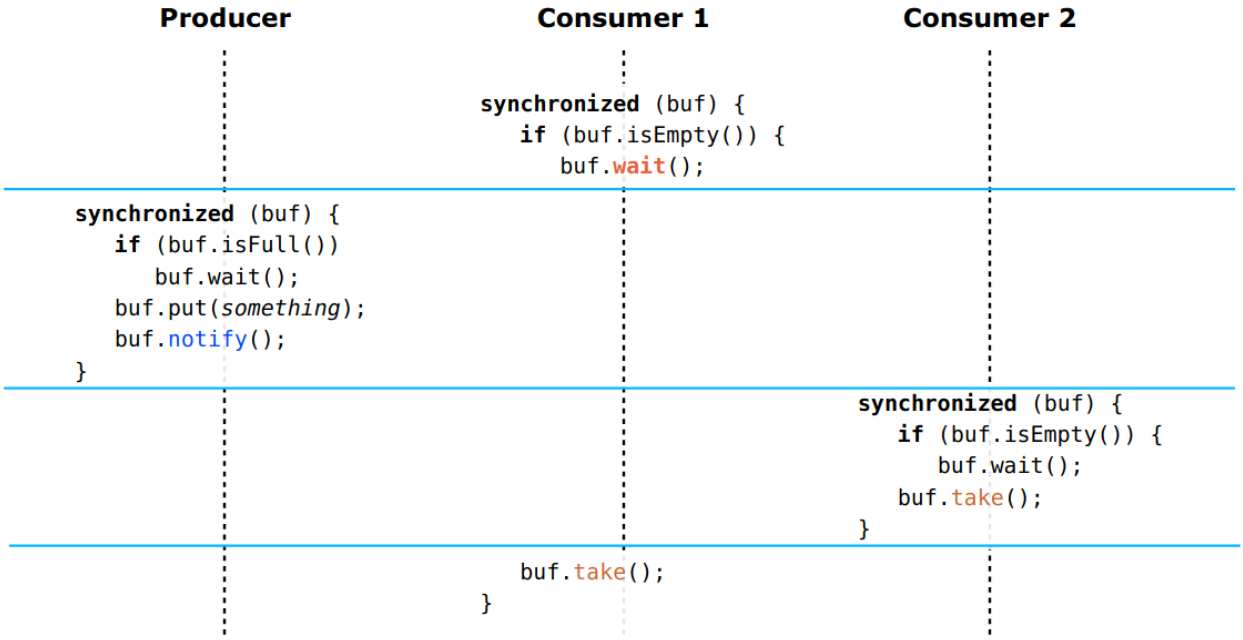
21.2 Supporto al multi-threading in Java



21.3 Problema 1: Consumatori multipli e notifyAll



Risultato: take da buffer vuoto!



Risultato: take da buffer vuoto!

21.4 Problema 2: Risvegli spuri

In casi eccezionali, il metodo wait può restituire il controllo al chiamante anche se non sono stati invocati i metodi notify e notifyAll

Questa eventualità è stata prevista per compatibilità con alcuni SO, nei quali le system call che la JVM utilizza per implementare le `condition variable` vengono interrotte in caso di segnali

Quindi, un consumatore che abbia trovato il buffer vuoto può uscire da `wait` senza un motivo specifico e tentare di leggere dal buffer vuoto.

Questo problema è risolvibile solo attraverso il controllo della condizione di attesa (attraverso un ciclo).

21.5 Perché utilizzare notifyAll

Conviene utilizzare `notifyAll` anche se ogni prodotto è destinato ad un unico consumatore.

Utilizzare notify **può creare un deadlock**.

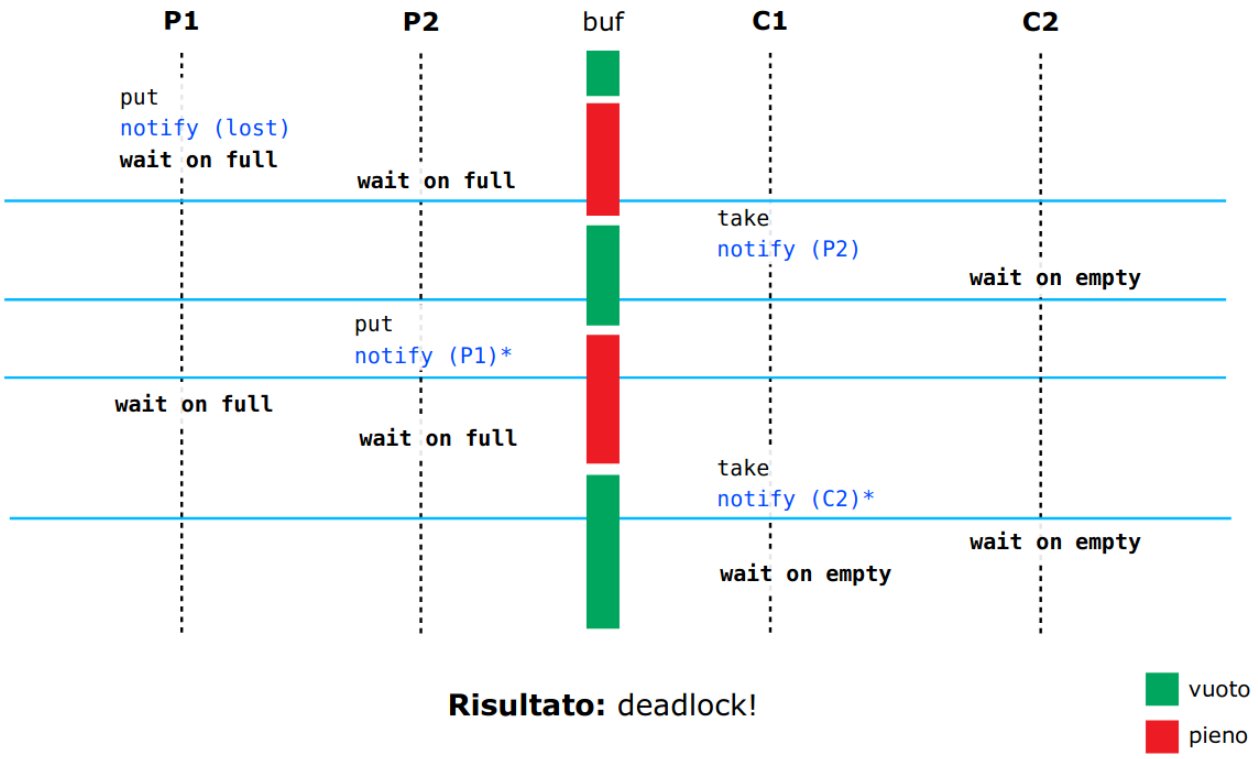
Schematizzando produttore e consumatore:

```
1  while (true){
2    wait on full
3    put
4    notify
5  }
```

```
1  while (true){
2    wait on empty
3    take
4    notify
5  }
```

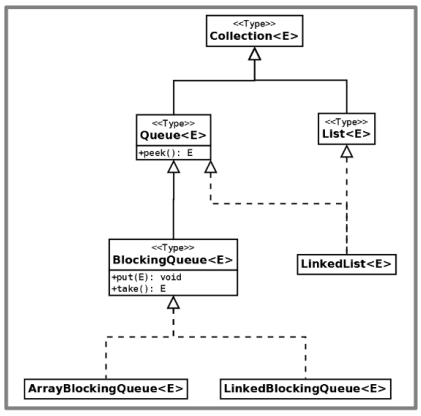
(Ogni notify è annotata con il thread che viene svegliato)

21.6 Deadlock dovuto a notify



21.7 Collezioni thread-safe: Code bloccanti

- Situazioni simili al produttore-consumatore sono così frequenti che la JCF offre delle collezioni predisposte all'utilizzo come buffer in un ambiente concorrente
- Parliamo delle code bloccanti;
- Di lato vengono presentate le principali interfacce e classi relative alle code bloccanti
  - Si parte dall'interfaccia parametrica **Queue**, che rappresenta una generica coda (non necessariamente bloccante);
  - Questa viene estesa da **BlockingQueue**, che rappresenta una generica coda bloccante;
  - Da questa otteniamo 2 implementazioni concrete



21.7.1 L'interfaccia Queue

Aggiunge alcuni metodi a Collection, tra i quali menzioniamo:

```
public E peek()
```

Che restituisce l'elemento in cima alla coda, senza rimuoverlo.

Se la coda è vuota, restituisce null.

Il metodo non è bloccante.

Siccome Queue estende Collection, tutte le code dispongono dei metodi offerti da Collection. Inoltre estende indirettamente Iterable.

Si noti che LinkedList implementa, oltre a List, anche Queue, ma non BlockingQueue.

21.7.2 L'interfaccia BlockingQueue

Rappresenta una generica coda bloccante.

Offre metodi per inserire e rimuovere elementi, che si bloccano se la coda è piena o vuota (rispettivamente).

Così facendo, essa permette a produttori e consumatori di sincronizzarsi senza usare esplicitamente le apposite primitive (mutex e condition variable).

L'interfaccia BlockingQueue offre, tra gli altri, i metodi *put* e *take*:

```
public void put(E elem) throws InterruptedException
```

- Inserisce l'oggetto `elem` nella coda;
- Le implementazioni scelgono in che posto inserirlo, tipicamente all'ultimo posto (ordine FIFO);
- Se la coda è piena, questo metodo mette il thread concorrente in **attesa** che venga rimosso qualche elemento dalla coda;
- Come tutti i metodi **bloccanti**, put è sensibile allo stato di interruzione del thread corrente, e solleva l'eccezione InterruptedException
  - Se tale stato diventa vero durante l'attesa;
  - Se era già vero all'inizio dell'attesa.

```
public void take(E elem) throws InterruptedException
```

- Restituisce e rimuove l'elemento in cima alla coda;
- Se la coda è vuota, questo metodo mette in attesa in thread corrente, finché non viene inserito almeno un elemento nella coda;
- Stesso ragionamento fatto per put vale per take.

21.7.3 Implementazioni delle BlockingQueue

La classe **ArrayBlockingQueue**, è una implementazione di BlockingQueue realizzata internamente tramite un array circolare.

Presenta capacità fissa, dichiarata una volta per tutte al costruttore:

```
public ArrayBlockingQueue(int capacity)
```

La classe **LinkedBlockingQueue** è un'altra implementazione di BlockingQueue, realizzata tramite una lista concatenata

- Una LinkedBlockingQueue ha una capacità potenzialmente illimitata
- Quindi, una LinkedBlockingQueue non risulta mai piena
- Pertanto, il metodo put di LinkedBlockingQueue non è bloccante
- Oltre ad essere bloccanti, queste classi sono anche thread-safe

Lo stesso non si può dire per le altre classi del Java Collection Framework, come LinkedList o HashSet, che possono dare risultati inattesi se utilizzate concorrentemente da più thread senza le opportune precauzioni di sincronizzazione.

I metodi put e take di queste due classi rispettano l'ordine FIFO

21.8 Produttore-consumatore con coda bloccante

- Le code bloccanti permettono di implementare in modo molto semplice lo schema produttore-

consumatore, senza doversi occupare manualmente della sincronizzazione

- I seguenti thread condividono il buffer

```
BlockingQueue<T> buffer = new ArrayBlockingQueue<T>(capacity);
```

Produttore e Consumatore

```
T x;
try {
  // attende se il buffer è pieno
  buffer.put(x);
} catch (InterruptedException e){
    return;
}
```

```
T x;
try {
  // attende se il buffer è vuoto
  x = buffer.take();
} catch (InterruptedException e){
    return;
}
// "consuma" x
```

21.9 I modelli di memoria

Un modello di memoria (memory model) è una descrizione di come una architettura hardware (reale o virtuale) gestisce gli accessi alla memoria.

Nelle architetture moderne la memoria è stratificata in diversi livelli, che vanno dalla memoria di massa, fino ai registri della CPU, passando per diversi livelli di cache:

- Questa stratificazione prendere il nome di *gerarchia della memoria* (memory hierarchy).

Alcuni livelli, come i registri e i primi livelli di cache, sono separati tra i diversi core;

Altri livelli (come la RAM) sono condivisi tra i core.

Questa stratificazione crea notevoli problemi di sincronizzazione tra processori diversi → Una famiglia di problemi di questo tipo prende il nome di *coerenza della cache*.

l memory model presenta un modello astratto del comportamento della memoria, in modo da consentire agli utenti (in primo luogo, sviluppatori di compilatori, di sistemi operativi, …) di ragionare senza conoscere i dettagli dell'hardware.

21.10 Il Java Memory Model

È una descrizione di come la JVM gestisce l'accesso alla memoria

È fondamentale in un contesto multi-threading.

Contiene regole di 3 tipi:

- **Atomicità** → Quali operazioni sono naturalmente atomiche?
- **Visibilità** → Quando una scrittura è visibile agli altri thread;
- **Ordinamento** → In che ordine vengono effettuate le operazioni?

Il JMM è un dei punti cardine della portabilità di Java → Offre alle applicazioni delle regole certe sull'accesso concorrente alla memoria.

22 Lezione del 01-06

22.1 Regole di Atomicità

22.1.1 Il modificatore volatile

`volatile` è un modificatore che si può applicare esclusivamente ai **campi** di una classe.

Indica che quel campo può essere modificabile da più thread.

Ha conseguenze di **atomicità**, **visibilità**, e **ordinamento**.

Un campo *volatile* non può essere *final*, in quanto l'accoppiata è priva di senso.

22.1.2 Regola di validità di operazioni naturalmente atomiche

Per atomiche intendiamo anche in assenza di meccanismi di mutua esclusione.

Regole

Sono **operazioni atomiche**:

1. La lettura e la scrittura di variabili di tipo primitivo, esclusi i tipi *long* e *double* e di tipo riferimento;
2. La lettura e la scrittura di variabili *volatili*

NB: La modifica di una variabile *long* può avvenire in 2 operazioni distinte, che modificano separatamente i 32 bit più significativi e i 32 bit meno significativi → Queste 2 operazioni potrebbero essere interrotte dallo scheduler.

22.1.3 Esempio

```
// Date le seguenti variabili:

int x, y;
long n;
volatile long m;
Object a, b;
volatile Object c, d;


// Esaminiamo le seguenti assegnazioni:

x = 8; // Operazione atomica

/*
  Questa operazione non è atomica, perchè comprende una lettura e una scrittura.
  Se l'operazione viene interrotta un altro thread può modificare `y` ed `x` potrebbe comunque
  assumere il vecchio valore di `y`
*/
x = y;

/*
```

```
    Questa operazione non è atomica
    Se l'operazione viene interrotta un thread potrebbe visualizzare
        n = 0x112233440000000000
    e un altro thread potrebbe invece visualizzare
        n = 0x0000000055667788
*/
n = 0x1122334455667788; // Costante long espressa in esadecimale

m = 0x1122334455667788; // Operazione atomica


a = null; // Operazione atomica

a = b; // Operazione non atomica per lo stesso motivo del caso 2

c = d; // Operazione non atomica per lo stesso motivo del caso 2 e 6
```

22.2 Regole di Visibilità

22.2.1 Analisi di un problema di visibilità

```
static boolean done;
static int n;

pubic static void main(String args[]){
    Thread T = new Thread(){
            public void run(){
                n = 42;
                try { sleep(1000); }
                catch (InterruptedException e) { return; }
                System.out.println("Fatto");
                done = true;
            }
    };
    t.start();
    while(!done) { } // Cattiva implementazione con attesa attiva
    System.out.println(n);
}
```

È importante notare che i 2 thread condividono le variabili *n* e *done*, ma non usano alcun meccanismo di sincronizzazione.

Il ciclo *while* del thread 1 può comportarsi come un ciclo infinito, anche se il thread 2 dopo un'attesa di un secondo esegue *done = true*.

Questo perché in mancanza di sincronizzazione, il *JMM* non offre alcuna garanzia su quando la scrittura nella variabile *done*

effettuata dal thread 2 sarà visibile al thread 1.

22.2.2 Principi fondamentali

- In mancanza di sincronizzazione, le operazioni (scritture in memoria) svolte da un thread possono rimanere nascoste agli altri thread a **tempo indefinito**;
- Alcune operazioni possono rimanere nascoste ed altre essere visibili.

La visibilità è garantita solamente dalle seguenti operazioni:

- Acquisire un **monitor** (cioè, entrare in un metodo o blocco sincronizzato) rende visibili le operazioni effettuate dall'ultimo thread che possedeva questo monitor, fino al momento in cui l'ha rilasciato;
- Leggere il valore di una variabile `volatile` **rende visibili** le operazioni effettuate dall'ultimo thread che ha modificato quella variabile, fino al momento in cui l'ha modificata;
- Invocare `t.start()` rende visibili al nuovo thread `t` tutte le operazioni effettuate dal thread chiamante, fino all'invocazione a `start()`;
- Ritornare da un'invocazione `t.join()` **rende visibili** tutte le operazioni effettuate dal thread `t` fino alla sua terminazione.

22.2.3 Rivisitazione dell'esempio precedente

```
1  // Modifichiamo la variable done con la keyword `volatile`
2  static volatile boolean done;
3  static int n;
4
5  pubic static void main(String args[]){
6      Thread T = new Thread(){
7              public void run(){
8                  n = 42;
9                  try { sleep(1000); }
10                 catch (InterruptedException e) { return; }
11                 System.out.println("Fatto");
12                 done = true;
13             }
14     };
15     t.start();
16     while(!done) { } // Cattiva implementazione con attesa attiva
17     System.out.println(n);
18 }
```

Il programma in questo modo avrà il comportamento atteso, perché ogni lettura della variabile `done` effettuata dal thread principale rende visibili le modifiche a `done` fatto dall'altro thread.

22.3 Confronto tra synchronized e volatile

Entrambe le keyword offrono garanzie di `atomicità` e `visibilità`.

Il modificatore *volatile* rende atomica soltanto una singola scrittura nella variabile in questione → Non rende un'assegnazione del tipo *a = b*, anche se entrambe *volatile*.

NB: Il modificatore *volatile* non rende atomica l'espressione `n++`.

Pertanto, un blocco o metodo `synchronized` rappresenta l'unica opzione per rendere atomica una sequenza di istruzioni (o comunque, per renderla mutuamente esclusiva rispetto ad altre sequenze critiche).

Il modificatore `volatile` è indicato nei casi in cui il contesto richieda la visibilità dei cambiamenti, ma non l'atomicità o la mutua esclusione.

22.4 Regole di Ordinamento

Si considerino i seguenti thread, che condividono 2 variabili `A` e `B`, inizialmente poste a `0`

Thread 1	Thread 2
`int r1;`	`int t2;`
`r1 = B;`	`r2 = A;`
`A = 1;`	`B = 1;`

- `r1 = 0, r2 = 1` se il Thread 1 viene eseguito per primo;
- `r1 = 1, r2 = 0` se il Thread 2 viene eseguito per primo;
- `r1 = 0, r2 = 0` se lo scheduler interrompe il primo thread tra le due assegnazioni;

In particolare, il JMM consente anche questo risultato:

`r1 = 1, r2 = 1`

Inoltre, in mancanza di sincronizzazione, al `compilatore/JVM/CPU` è consentito di riordinare le istruzioni, a patto che tale riordino sia ininfluente dal punto di vista del singolo thread.

In questo caso, è consentito invertire l'ordine delle 2 istruzioni del Thread 1 (o 2).

Se l'ordine viene invertito il risultato precedente diventa possibile.

In generale, il compilatore e la JVM possono riordinare qualsiasi sequenza di istruzioni, a patto che il risultato non cambi per un singolo thread che esegua quelle istruzioni.

I costrutti *synchronized* e *volatile* riducono le possibilità di riordino

Consideriamo 2 istruzioni `x` e `y` successive. La tabella sottostante specifica se è possibile **scambiare** di posto le 2 istruzioni:

tipo di x \ tipo di y	normale	lettura volatile inizio synchronized	scrittura volatile fine synchronized
normale	**Si**	**Si**	No
lettura volatile inizio synchronized	No	No	No
scrittura volatile fine synchronized	**Si**	No	No

In pratica:

1. Le istruzioni normali si possono sempre scambiare;
2. Le istruzioni normali si possono portare *dentro* un blocco sincronizzato;
3. Le istruzioni normali che precedono la lettura di una *volatile* si possono spostare dopo la lettura
4. Le istruzioni normali che seguono la scrittura di una *volatile* si possono spostare dopo la scrittura;

22.5 Esempio con blocchi sincronizzati

Introduciamo un oggetto condiviso *obj* e 2 blocchi sincronizzati

Thread 1 e Thread 2

```
1  int r1;
2  synchronized (obj) {
3    r1 = B;
4    A = 1;
5  }
```

```
1  int r2;
2  synchronized (obj) {
3    r2 = A;
4    B = 1;
5  }
```

I blocchi sincronizzati non impediscono alle istruzioni al loro interno di essere riordinate, ma rendono i 2 blocchi mutuamente esclusivi.

Quindi, uno dei due blocchi verrà interamente eseguito prima dell'altro.

Gli unici output possibili sono:

- `r1 = 0, r2 = 1;`
- `r1 = 1, r2 = 0;`

22.6 Esempio con volatile

Supponiamo adesso che `A` e `B` siano *volatile*.

Consultando le regole di ordinamento, scopriamo che il compilatore adesso non può riordinare le istruzioni, perché sono tutte letture o scritture di variabili volatile.

Gli output possibili sono:

- `r1 = 0, r2 = 0;`
- `r1 = 0, r2 = 1;`
- `r1 = 1, r2 = 0;`

Il primo output può capitare perché volatile non rende i due thread mutuamente esclusivi.

23 Lezione del 06-06

23.1 Mutua atomicità

Due operazioni (o blocco di istruzioni) `A` e `B`, sono mutualmente atomiche se, dal punto di vista di un thread che sta eseguendo A, gli effetti dell'esecuzione di B da parte di un altro thread vengono visiti per intero, o per niente (ma mai a metà).

Questa è una forma più debole di atomicità, **relativa** anziché **assoluta**

È una forma più astratta di mutua esclusione → Il termine suggerisce che le 2 operazioni verranno eseguite in tempi diversi (non contemporaneamente).

Il termine mutua atomicità parla solo di visibilità degli effetti, non di implementazione.

La mutua esclusione è un *modo di implementare la mutua atomicità*.

La mutua atomicità viene garantita dai meccanismi di locking (come il costrutto synchronized)

Tutti i blocchi synchronized sullo stesso monitor sono mutualmente atomici

23.2 Lazy initialization

Consideriamo il problema di una classe che voglia rendere disponibile un oggetto, che sarà istanziato soltanto alla prima richiesta. Questo problema prende il nome di *lazy inizialization*.

Una soluzione naif è la seguente

```
// Il riferimento special sarà inizializzato con un nuovo oggetto alla prima invocazione di
    getSpecial.
class A {
    private static HeavyClass special;
    public static HeavyClass getSpecial(){
        if (special == null)
            special = new HeavyClass();
        return special;
    }
}
```

Sfortunatamente questa operazione non è thread-safe

23.2.1 Primo problema della lazy initialization

Scenario 1

Due thread invocano contemporaneamente `getSpecial()`:

- Il primo thread trova special a *null* e viene interrotto dallo scheduler. Anche il secondo trova special a *null*, quindi istanzia un oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a *special* e esce dal metodo;
- Il primo thread riprende la sua esecuzione, istanzia un secondo oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a *special* ed esce dal metodo;

NB: Sono stati istanziati 2 diversi oggetti HeavyClass, contrariamente alle intenzione. Questo è il classico problema dovuto alla mancata atomicità della sequenza *lettura di special* → *scrittura di special* (Questo problema non è legato alle sottigliezze del JMM)

23.2.2 Secondo problema della lazy initialization

Scenario 2

Due thread invocano contemporaneamente `getSpecial()`:

- Il primo thread trova special a *null* quindi istanzia un oggetto di tipo *HeavyClass*, ne assegna l'indirizzo a *special*;
- Il secondo thread riceve dal metodo un riferimento allo stesso oggetto, accede a questo oggetto e lo trova in uno stato incoerente (non completamente inizializzato dal costruttore).

  NB: 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.

In mancanza di sincronizzazione, non vi è alcuna garanzia che il secondo thread vede tutte le operazioni svolte dal primo. Potrebbe darsi che il secondo thread veda il valore corrente di special (cioè l'indirizzo del nuovo oggetto *HeavyClass*), ma non veda il valore corrente di alcuni campi di quel'oggetto.

23.2.3 Soluzione al problema della lazy initialization

Dichiarare sincronizzato (`synchronized`) il metodo `getSpecial()` risolve entrambi i problemi descritti.

Infatti il primo problema viene risolto rendendo mutualmente esclusive le invocazioni a `getSpecial`.

Il secondo problema viene risolto grazie alle garanzie di visibilità che offre la keyword `synchronized`.

Questa soluzione impone però un **overhead di performance**, dovuto alla necessità di `acquisire/rilasciare` un mutex, su tutte le invocazioni di `getSpecial()`, anche molto tempo dopo l'inizializzazione dell'oggetto `HeavyClass`, quando ormai la sincronizzazione non sarebbe più necessaria

23.2.4 Soluzione avanzata al problema della lazy initialization

```
class A {
    private static class HeavyClassHolder {
        // Il riferimento special viene spostato all'interno di una classe interna statica e privata.
        static HeavyClass special = new HeavyClass();
    }

    public static HeavyClass getSpecial(){
        return HeavyClassHolder.special;
    }
}
```

In questo modo la classe `HeavyClass` sarà istanziata quando il campo statico *special*, verrà istanziato.

La `VM` inizializza la classe *HeavyClassHolder*, e quindi il suo campo *special*, soltanto al **primo utilizzo**, cioè alla prima invocazione di `getSpecial()`.

Non ci sono problemi di sincronizzazione perché la `VM` garantisce che l' **inizializzazione di una classe sia un'operazione atomica**:

```
class A {
    private static B b = <inizializzatore>;
    static {
        <blocco di inizializzazione statico>
    }
}
```

Con questa modalità si risolvono i problemi della lazy initialization, senza dover ricorrere ad una sincronizzazione esplicita.

24 Lezione del 10-06 (lezione finale)

Un `method reference` è una espressione che denota un metodo (I puntatori a funzione arrivano in questo modo in Java).

Molto simile ai puntatori a funzione in `C/C++`.

Risulta molto più efficiente rispetto alla riflessione

24.1 Forme

- Metodi statici → `Employee::getMaxSalary`;
- Metodi di istanza, istanza non specificata → `Employee::getSalary`;
- Metodi di istanza, istanza specificata → `mike::getSalary`;
- Costruttori → `Employee::new`;

Tra quelli poco comuni:

- Metodi di istanza della super classe → `super::foo`;
- Array Constructor → `A[]::new`.

24.2 Type Inference nei `method reference`

Questo tipo di reference non ha un tipo intrinseco → La type inference assegna un tipo in base al contesto → Questo rende obbligatorio il contesto (che deve identificare una `Interfaccia funzionale`).

24.2.1 Contesti validi per i `method reference`

Gli stessi per le lambda:

- RHS di un assegnamento;
- Parametri attuali del metodo o del costruttore;
- Argomento del `return`;
- Argomento del `cast`

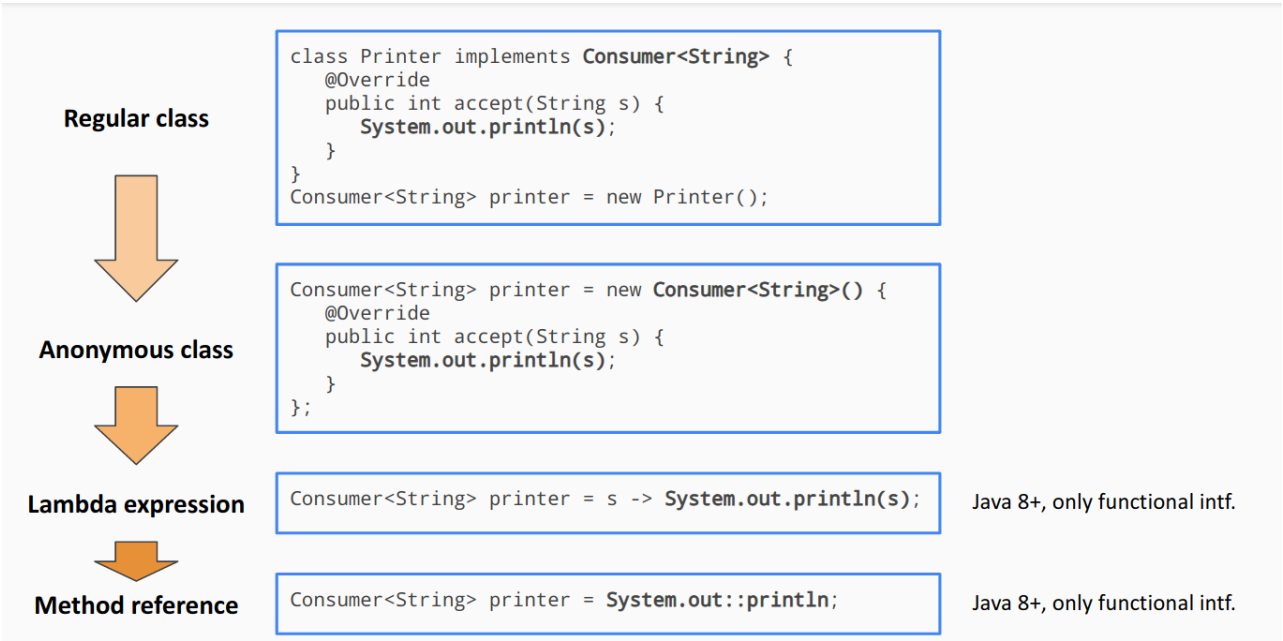
Il tipo ricevuto `T` deve essere una Interfaccia funzionale:

Contesto	Esempio	Contesto ricevuto
Assegnamento di RHS	`T var = <method ref>`	`T`
Parametro attuale di un metodo o costruttore	`foo(<method ref>)`	Tipo del corrispondente parametro formale
Argomento del `return`	`return <method ref>`	Tipo di ritorno del metodo corrente
Argomento del `cast`	`(T) <method ref>`	`T`

Sia le lambda espressioni che i metodi reference sono una modalità comoda di passare codice in giro:

- Le lambda per *snippet one-shot*;
- I method reference per casi *più generici*;

24.3 Dalle classi a metodi reference



24.4 Interfaccia FI

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

24.5 Riferimenti

- Functional Java