

Programmazione Orientata agli Oggetti

Classi Nidificate

Sommario

- Introduzione
- Nidificazione di Classi (non statica)
 - Interne
 - Locali
 - Anonime
- Nidificazione Statica
 - Classi
 - Interfacce
- Conclusioni

Classi Nidificate (o «Interne»)

- Disponibili a partire da Java 1.1 (non da subito, quindi)
- All'epoca motivate dalla necessità di facilitare la scrittura di codice per la gestione di «eventi grafici»
 - Il fatto che questa esigenza si sia avvertita, forte, anche dopo Java 1.1, testimonia forse il non totale successo dell'iniziativa in tal senso
 - Difatti le stesse esigenze (ed anche altre) hanno motivato ulteriori e successive iniziative
 - le lambda function di Java 8
- Rimangono comunque diffuse, e ben motivate, anche in altri contesti di utilizzo, come ulteriore strumento per la definizione di tipi

Definizione di Classi Nidificate

- Una classe nidificata è una classe dichiarata all'interno del corpo della definizione di un'altra classe
 - la classe dichiarata all'interno è detta classe «interna»
 - la classe dichiarata all'esterno è detta classe «esterna»

```
public class ClasseEsterna {  
    ...  
    class ClasseInterna {  
        ...  
    }  
}
```

Quattro Varianti per «Nidificare»

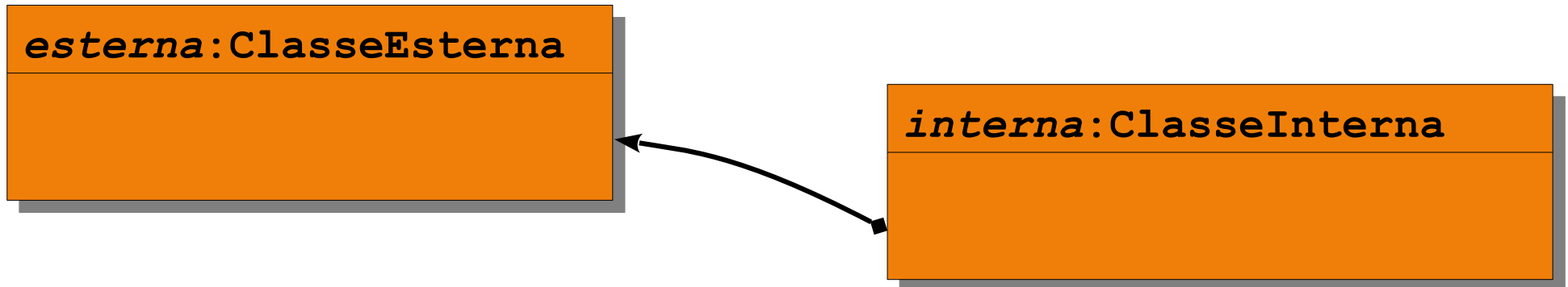
- Esistono quattro diverse varianti di *classi nidificate*:
 - *Interne*
 - ad es. usate per le classi degli iteratori dipendenti dal tipo di collezione all'interno della classe della collezione
 - *Locali*
 - dichiarate all'interno di un blocco di codice direttamente nel corpo di un metodo
 - ad es. un comparatore “usa e getta”
 - *Anonime*
 - consentono di *NON* specificare il nome della classe concreta
 - ad es. un comparatore “usa e getta”
- ...e per finire:
 - *Statiche*

Quattro Varianti; Statiche vs Non-Statiche (1)

- La semantica delle prime tre varianti è nettamente diversa rispetto a quella delle classi nidificate statiche
 - L'idea fondamentale alla loro base è che un'istanza della classe interna deve mantenere un *riferimento* all'istanza della classe esterna che l'ha creato
 - ✓ Questo *NON* vale solo per le classi nidificate statiche
- Le classi nidificate statiche si possono inquadrare, nella sostanza, quasi solo come ad un meccanismo di *namespacing*
- Rende esplicito ed evidente un legame esistente tra due classi, ma senza differenze concettuali sostanziali rispetto alle ordinarie classi non nidificate (>>)

Quattro Varianti; Statiche vs Non-Statiche (2)

- Invece per le tre prime varianti di classi nidificate, al legame di contenimento delle classi nel codice, corrisponde effettivamente un forte legame tra le istanze coinvolte a tempo dinamico
- Una istanza della classe interna è sempre associata ad una sola istanza della classe esterna
- L'istanza interna **non** può esistere senza una istanza della classe esterna a cui rimane permanentemente associata sin dalla creazione
- Per creare l'istanza interna serve quella esterna



Classe Nidificata Interna

- Una classe interna è una classe definita all'interno di un'altra classe (anche detta *esterna*)

```
public class Outer {  
    ...  
    public class Inner { ... }  
    ...  
}
```

- Le regole di visibilità sono simili a quelle dei metodi
 - Un oggetto della classe interna conserva sempre un riferimento all'oggetto della classe esterna a partire dal quale è stato istanziato
 - L'oggetto interno può accedere alle variabili di istanza e ai metodi della classe esterna, anche se privati
 - Se dichiarata privata non può essere invece utilizzata dall'esterno (della classe «esterna»)

Classe Interna: Esempio

```
public class Outer {  
    private String outerName = "outer";  
  
    public class Inner {  
        private String innerName = "inner";  
        public void saluta() {  
            return innerName + " " + outerName;  
        }  
    }  
}
```

Classe Interna ed Esterna: Legame tra le Istanze (1)

- Ogni istanza di una classe interna è legata ad un singolo oggetto della classe esterna in cui viene dichiarata
 - Non può esistere un oggetto «interno» (ovvero istanza della classe interna) senza un oggetto «esterno» (istanza della classe esterna) che lo crei e verso cui conserva un riferimento sin dalla sua creazione (e fino alla sua deallocazione)
- Nell'esempio precedente un metodo della classe nidificata **Inner** accede alla variabile **outerName**
 - Questa è una variabile di istanza appartenente all'oggetto «esterno»

Classe Interna ed Esterna: Legame tra le Istanze (2)

- La sintassi da utilizzarsi per creare istanze della classe interna rende evidente l'esistenza di questo legame

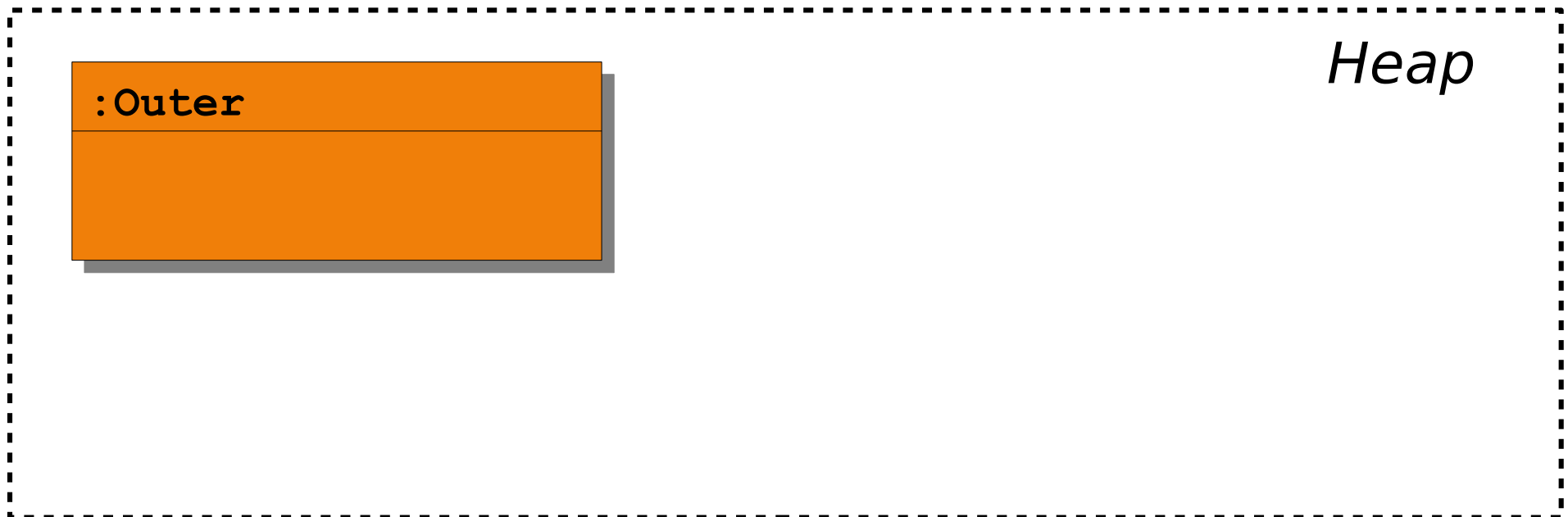
```
public class InnerTest {  
  
    @Test  
    public void testSaluta() {  
        Outer outer = new Outer();  
        Outer.Inner inner = outer.new Inner();  
        assertEquals("inner outer", inner.saluta());  
    }  
}
```

- Possibile solo se la classe interna *non* è dichiarata privata
- N.B.
 - In questo esempio l'oggetto «interno» viene creato dall'esterno della classe **Outer** che include al suo interno la definizione di **Inner**


Classi Interne (1)

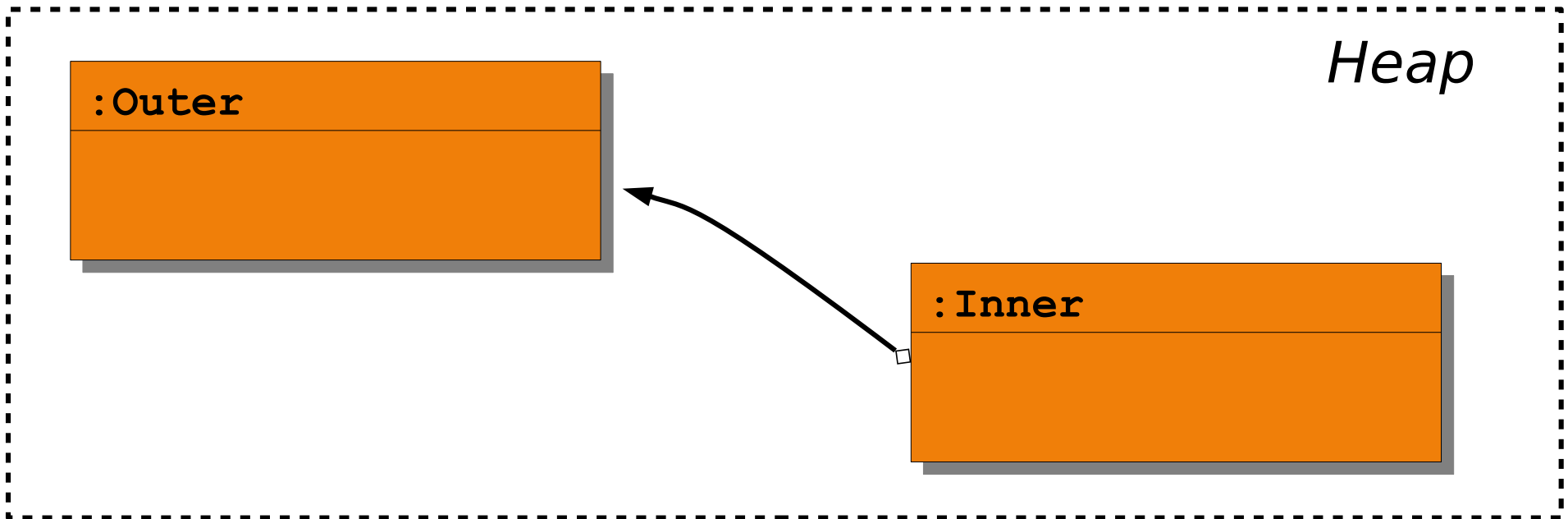
```
@Test
public void testSaluta() {
    Outer outer = new Outer();
    ▶ Outer.Inner inner = outer.new Inner();

    assertEquals("inner outer", inner.saluta());
}
```



Classi Interne (2)

```
@Test
public void testSaluta() {
    Outer outer = new Outer();
    Outer.Inner inner = outer.new Inner();
    
    assertEquals("inner outer", inner.saluta());
}
```



Creazioni di Istanze Nidificate dall'Interno

- All'interno di una classe si può creare un'istanza di una classe nidificata come se si trattasse di una qualsiasi classe ordinaria:

```
public class Outer {  
    ...  
    public void riproponiSaluto() {  
        System.out.println("La inner saluta con:");  
        Inner inner = new Inner();  
        inner.saluta();  
    }  
}
```

- In questo caso l'istanza della classe esterna **Outer** referenziata dalla nuova istanza della classe nidificata **Inner** sarà quella esterna che ha effettuato l'invocazione del costruttore della classe interna (ovvero l'istanza il cui riferimento, prima dell'invocazione, è **this**)

Accesso alle Variabili di Istanza nelle Classi Nidificate

- La classe interna può accedere alle variabili di istanza (anche se private) della classe esterna
 - `Inner` accede alla variabile di istanza `outerName` della classe `Outer`

```
public class Outer {  
    private String outerName = "outer";  
    public class Inner {  
        private String innerName = "inner";  
        public void saluta() {  
            System.out.println(innerName + " " +  
                                outerName);  
        }  
    }  
}
```

Shadowing e Classi Nidificate (1)

- Se la classe interna dichiara una variabile (di istanza o locale) con lo stesso nome di una variabile di istanza della classe esterna, quest'ultima risulterà *non* più accessibile:
 - la nuova variabile (di istanza o locale) ha uno *scope* più ristretto e “mette in ombra” la variabile di istanza della classe esterna
 - Per poter accedervi bisogna qualificare completamente il nome con la sintassi

`<Nome_OuterClass>.this.<Nome_Variabile>`

- Ad esempio:

`Outer.this.name;`

Shadowing e Classi Nidificate (2)

```
public class Outer {  
    private String name = "outer";  
    public class Inner {  
        private String name = "inner";  
        public void saluta() {  
            System.out.println(name + " " +  
                                Outer.this.name) ;  
        }  
    }  
}
```

Accesso alle Variabili di istanza della Classe Esterna

- La sintassi

`<Nome_OuterClass>.this.<Nome_Variabile>`

sebbene decisamente prolissa, risulta efficace:

- rende chiaro a quale variabile si fa riferimento
- simile a quanto già detto per `this` con riferimento alle variabili di istanza

Esempio di Classi Nidificate Interne: `Iterator<E>` ed `ArrayList<E>`

- `ArrayList<E>` mette a disposizione un iteratore per enumerare gli elementi che contiene
- Un iteratore è un oggetto istanza di una classe sottotipo (concreto) di `Iterator<E>`
 - Per poter fornire gli elementi di una `ArrayList<E>`, l'iteratore deve poter accedere al sottostante array che li contiene
 - tale array è privato e quindi, per potervi accedere, l'implementazione dell'iteratore mediante una classe interna ad `ArrayList<E>` risulta ben motivata

ArrayList<E>, Vista dall'Interno (1)

- **ArrayList<E>** gestisce i dati al suo interno con un array di **Object**:

```
public class ArrayList<E> extends AbstractList<E> implements
    List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    // Dati dell'ArrayList
    private transient Object[] elementData; // Gli elementi

    private int size; // Dimensione dell'array

    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                              initialCapacity);
        this.elementData = new Object[initialCapacity];
    }
    ...
}
```

ArrayList<E>, Vista dall'Interno (2)

- Gli elementi sono conservati in un array di `Object`, tuttavia viene sempre effettuato un *cast* esplicito nei metodi `get` (dell'interfaccia `List<E>`)

```
public E get(int index) {  
    rangeCheck(index); // index < size ?  
  
    return elementData(index);  
}
```

```
E elementData(int index) {  
    return (E) elementData[index];  
}
```

`ArrayList<E>: Iterator<E>`

- `ArrayList<E>` implementa (indirettamente) l'interfaccia `Iterable<E>` e possiede dunque un metodo

```
public Iterator<E> iterator()
```

che restituisce un iteratore, istanza della classe `Itr<E>`

- `Itr` è una classe interna di `ArrayList<E>`
- `Itr` implementa `Iterator<E>`

- Ecco il metodo `iterator()` che viene utilizzato per ottenere un iteratore:

```
public Iterator<E> iterator() {  
    return new Itr();  
}
```

La Classe Interna Itr (1)

- La classe `Itr` implementa i metodi di `Iterator<E>`:
 - `public E next()`
 - `public void remove()`
 - `public boolean hasNext()`

```
// All'interno di ArrayList ...
```

```
private class Itr implements Iterator<E> {  
    int cursor;    // = 0  
    int lastRet = -1;  
    int expectedModCount = modCount;  
    ...  
}
```

La Classe Interna Itr (2)

```
private class Itr implements Iterator<E> {  
    int cursor;  
    int lastRet = -1;  
    int expectedModCount = modCount;  
    ...  
}
```

- **int cursor**
 - L'indice del prossimo elemento che **next()** deve restituire
- **int lastRet**
 - L'indice dell'ultimo elemento restituito da **next()**

La Classe Interna Itr: next()

```
private class Itr implements Iterator<E> {

    public E next() {
        checkForComodification();
        int i = cursor;
        if (i >= size)
            throw new NoSuchElementException();
        Object[] elementData = ArrayList.this.elementData;

        if (i >= elementData.length)
            throw new ConcurrentModificationException();

        cursor = i + 1;
        return (E) elementData[lastRet = i];
    }
}
```

La Classe Interna `Itr`: Altri Metodi

- La classe `Itr` dispone di altri metodi
- `hasNext()`
 - Restituisce: `cursor != size`

`size` è una variabile di istanza privata di `ArrayList` a cui però è possibile accedere da una classe interna
- `remove()`
 - Rimuove l'ultimo elemento restituito dall'iteratore usando i metodi di `ArrayList` e `lastRet`

Vantaggi delle Classi Interne

- Usando una classe interna non è necessario che **ArrayList<E>** esponga la propria rappresentazione interna, e specifica, degli *iteratori*
- Questo permette di applicare più agevolmente i principi *dell'information hiding*:
 - In caso contrario sarebbe necessario un metodo getter per ottenere l'accesso all'array interno (dentro) **ArrayList**
- Anche per la distribuzione delle responsabilità la scelta è opportuna
 - L'iteratore e la collezione possiedono responsabilità ben definite e distinte (sebbene strettamente correlate) ed è quindi desiderabile definirle in due classi distinte
- Per *economia di pensiero*: non è nemmeno necessario conoscere il nome del tipo dinamico degli iteratori delle collezioni, ma solo il supertipo astratto **Iterator<E>**

Utilizzi delle Classi Interne

- Raggruppare logicamente classi che vengono utilizzate assieme senza essere costretti a renderle tutte visibili esternamente
- Favorire l'incapsulamento: una classe interna può accedere anche ai dati privati della classe esterna
 - Non è quindi necessario avere getter e setter per esporre la rappresentazione dati nelle variabili di istanza private anche quando si vuole condividere tale rappresentazione
- Favorire la coesione: tenendo la dichiarazione di una classe interna ben vicina ai suoi unici utilizzi (da parte della classe esterna che la definisce), si favorisce la qualità del codice

Classi Nidificate Locali (1)

- Una classe locale è una classe definita all'interno di un blocco di codice (solitamente il corpo di un metodo)
- Il suo campo d'azione (scope), come per una variabile locale, è il blocco di codice all'interno del quale è viene dichiarata:

```
{  
    class LocalInner {  
        ...  
    }  
    LocalInner inner = new LocalInner();  
}
```

// Qui non si può usare LocalInner

Classi Nidificate Locali (2)

- Le classi locali possiedono le stesse caratteristiche delle classi interne
 - Possono essere create solo a partire da un'istanza di una classe esterna e mantengono sempre un riferimento a tale istanza tramite `<Nome_OuterClass>.this`
 - L'istanza esterna è quella che sta eseguendo il blocco di codice contenente la dichiarazione della classe locale
- ✓ Le classi locali non possono mai essere istanziate dall'esterno (per costruzione!)

Classi Locali: Esempio (1)

- Le classi locali spesso servono ad utilizzi “usa-e-getta”, all'interno di un unico metodo
 - In questo modo è possibile avere tutto il codice in un unico punto rendendolo più leggibile e vicino all'unico utilizzo
- Un classico esempio prevede la definizione di comparatori
 - E' veramente necessario creare una nuova classe (in un ennesimo file) per un comparatore mono-uso?
 - Ad es. perché serve un ordinamento particolare per fare una stampa ordinata in un solo punto del codice

Classi Locali: Esempio (2)

```
class Aula {  
    private List<Studente> studenti;  
  
    ...  
  
    public List<Studente> ordinaStudentiPerEta() {  
        class ComparatorePerEta implements Comparator<Studente> {  
            @Override  
            public int compare(Studente s1, Studente s2) {  
                return s1.getEta() - s2.getEta();  
            }  
        }  
  
        ComparatorePerEta comp = new ComparatorePerEta();  
        List<Studente> studentiOrdinati = new ArrayList<>(studenti);  
        Collections.sort(studentiOrdinati, comp);  
        return studentiOrdinati;  
    }  
}
```


Classi Locali & Variabili Locali

- Le classi locali possono accedere alle variabili locali del metodo (in generale, del blocco di codice) ove definite
 - È però richiesto che tali variabili locali siano dichiarate `final` (Java 7-) o che siano *effectively final* (Java 8+)

<https://stackoverflow.com/questions/4732544/why-are-only-final-variables-accessible-in-anonymous-class>

- Quando una classe locale fa riferimento ad una variabile locale, una copia del valore di tale variabile viene “catturata”
 - ✓ Subito: nel momento della creazione dell’istanza nidificata
 - ✓ ma potrà essere usata anche successivamente
 - ✓ anche se il ciclo di vita della copia *originale* termina
- Non potrà accedere (e tantomeno modificare) l’originale ma *solo* la copia
 - Il vincolo sintattico sulle variabili `final` rende tutto questo palese

Classi Locali & Variabili Locali: Esempio

```
class Canale {  
    private List<Studente> studenti;  
  
    ...  
  
    public SortedSet<Studente> ordinaStudentiPerVoto(  
        final Map<Studente, Integer> studente2voto) {  
        class ComparatorePerVoto implements Comparator<Studente> {  
            @Override  
            public int compare(Studente s1, Studente s2){  
                int votoS1 = studente2voto.get(s1);  
                int votoS2 = studente2voto.get(s2);  
                return ( votoS1 - votoS2 ) ;  
            }  
        }  
  
        ComparatorePerVoto comp = new ComparatorePerVoto();  
        SortedSet<Studente> studentiOrdinati = new TreeSet<>(comp);  
        studentiOrdinati.addAll(studentiOrdinati);  
        return studentiOrdinati;  
    }  
}
```

Classi Locali & Variabili Locali: Esempio (cont.)

- Nel precedente esempio una collezione di studenti viene ordinata sulla base del voto
 - Il metodo `ordinaPerVoto()` riceve una mappa che associa gli studenti al voto da loro conseguito
- Tale mappa può essere acceduta direttamente all'interno del comparatore
 - deve però essere esplicitamente dichiarata come **final**
 - Il riferimento è costante (il contenuto della mappa non necessariamente)

Classi Nidificate Anonime

- Le classi nidificate locali spesso sono classi usa-e-getta
 - unico utilizzo, o cmq utilizzi confinati all'interno del blocco contenente la definizione
 - comode se risulta eccessivo definire un'intera nuova classe solo per utilizzi "locali"
- Talvolta può risultare eccessivo anche solo dover scegliere un nome per una classe mono-uso
 - soprattutto se già si conosce il tipo astratto (l'interface) che dovrà implementare
- Le classi anonime permettono di definire ed istanziare una classe in un'unica espressione
 - risultano molto utili quando di tale classe è necessaria un'unica istanza per un unico utilizzo

Classi Anonime: Esempio

- Le classi anonime vengono definite sempre e comunque per estensione/implementazione di un supertipo già noto
 - Un'interfaccia (viene implementata)
 - Una classe astratta (viene estesa)
 - Una superclasse (viene estesa)

```
Comparator<Studente> comp = new Comparator<Studente>() {  
    @Override  
    public int compare(Studente s1, Studente s2) {  
        ...  
    }  
}; // N.B. ';' necessaria
```

Classi Anonime ed Estensione

- La creazione di una istanza di una classe anonima avviene sempre con **new** seguito dal nome di un'interfaccia, una classe o una classe astratta
- Come nelle ordinarie estensioni di classe/implementazioni di interfacce:
 - è possibile aggiungere nuove variabili di istanza e nuovi metodi
 - è possibile implementare metodi
 - tutti quelli previsti dall'interfaccia
 - tutti quelli abstract della superclasse astratta
 - è possibile sovrascrivere metodi
- La classe così definita è quindi, a tutti gli effetti, una nuova classe, senza nome (appunto, *anonima*)

Istanza di una Classe Anonima

- La definizione di una classe anonima e la creazione della sua unica istanza avviene contestualmente, mediante un'unica espressione

```
Comparator<Studente> comp = new Comparator<Studente> {  
    @Override  
    public int compare(Studente s1, Studente s2) {  
        ...  
    }  
}; // Notare ; → questa è un'espressione
```

- comp** conterrà un riferimento all'istanza della classe anonima appena definita
 - può esistere solo un'istanza, creata al momento stesso della definizione di classe

Classi Anonime vs Locali

- Le classi nidificate anonime sono sostanzialmente delle classi nidificate locali senza nome, possono quindi:
 - far riferimento all'oggetto della classe esterna dalla quale sono create mediante `<Nome_OuterClass>.this` e accedere ai suoi metodi e variabili di istanza private
 - accedere alle variabili visibili nel blocco di codice in cui vengono definite, purché siano dichiarate **final**

Classi Anonime: Costruttori

- Una classe anonima viene creata invocando uno dei costruttori della sua superclasse
 - direttamente il costruttore *no-args* di `java.lang.Object` (se non estende altre classi ma implementa una interface)
 - ✓ la classe viene definita ed istanziata contestualmente, non è quindi possibile definire nuovi costruttori e, contemporaneamente, invocarli
- É comunque possibile invocare i costruttori della superclasse (visibili) o catturare variabili locali **final**

Comparator<T> Anonimi

```
class Canale {  
    private List<Studente> studenti;  
  
    ...  
  
    public List<Studente> ordinaStudentiPerVoto(  
        final Map<Studente, Integer> studente2voto) {  
        List<Studente> studentiOrdinati = new ArrayList<>(studenti);  
        Collections.sort(studentiOrdinati, new Comparator<Studente>() {  
            @Override  
            public int compare(Studente s1, Studente s2) {  
                return studente2voto.get(s1) - studente2voto.get(s2);  
            }  
        });  
        return studentiOrdinati;  
    }  
}
```

Esempio Invocazione Costruttore

- Per testare il comportamento di una classe astratta (che non si può istanziare dirett.)

```
AbstractPersonaggio out =  
    new AbstractPersonaggio("fake") {  
        @Override  
        public String agisci(Partita partita) {  
            return "fake";  
        }  
    };
```

- La definizione di classe (nidificata anonima locale) è contestuale al suo primo ed unico istanziamento

Codice Oggetto (.class) per Classi Nidificate

- Dopo la compilazione i file `.class` generati per le classi nidificate risultano separati da quelli associati alle classi che le contengono
 - le classi interne vengono salvate in file di nome `<NomeOuter>$<NomeInner>.class`
 - per le classi anonime, si usa un numero progressivo `<NomeOuter>$1.class, <NomeOuter>$2.class ...`
 - Per le classi locali:
`<NomeOuter>$1<NomeLocal>.class,`
`<NomeOuter>$2<NomeLocal>.class ...`
- Quest'ultime possiedono un nome ma è comunque lecito dichiarare due classi locali omonime

Classi Nidificate e Riflessione

- I nomi dei file `.class` associati alle classi nidificate sono esattamente quelli (tranne l'estensione `.class`) ottenibili tramite l'introspezione:

```
public class Outer {
    class Inner {}
    public void inner() {
        Inner inner = new Inner(); // Stampa:
        System.out.println(inner.getClass().getName()); // 'Outer$Inner'
    }

    public void local() {
        class Local {}
        Local local = new Local();
        System.out.println(local.getClass().getName()); // 'Outer$1Local'
    }

    public void anonymous() {
        Object anonymous = new Object(){};
        System.out.println(anonymous.getClass().getName()); // 'Outer$1'
    }
}
```

Classi Statiche Nidificate (1)

- Le classi nidificate statiche sono classi dichiarate all'interno di altre classi usando il modificatore **static**:

```
public class Outer {  
    ...  
    public static class Nested {  
        ...  
    }  
}
```

Classi Statiche Nidificate (2)

- Le classi statiche, sono diverse da tutte le altre tipologie di classi nidificate viste sinora
- ✓ Le sue istanze *NON* sono create a partire da un oggetto della classe esterna
- Sono create come le classi ordinarie
 - sia dall'esterno della classe stessa:
`Outer.Nested nested = new Outer.Nested();`
 - sia dall'interno della classe (ma il nome completamente qualificato non è necessario):
`Nested nested = new Nested();`

Classi Statiche Nidificate (3)

- Pertanto un'istanza di una classe statica nidificata non mantiene alcun legame con gli oggetti della classe in cui viene dichiarata:
 - semplicemente è una classe con un nome dipendente dal nome della classe esterna
- Quanto già discusso per le altre tre tipologie di classi nidificate non si applica alle classi nidificate statiche
- Una classe statica può essere:
 - Privata: non può essere usata all'esterno della classe in cui viene dichiarata
 - Pubblica: può essere usata anche all'esterno della classe in cui viene dichiarata

Utilizzo delle Classi Statiche

- Possono essere usate quando non si ha la necessità di avere un riferimento ad un oggetto della classe esterna ma ha ancora senso mantenere le due classi (esterna vs interna) “vicine”
 - Ad es. perchè rappresentano concetti naturalmente ed imprescindibilmente molto legati
- Per un esempio semplice ma convincente, conviene però far riferimento anche alle interfacce nidificate, che sono usate già nel JCF
 - Rispetto al caso delle classi statiche nidificate, non sussistono differenze (specificatamente dovute alla nidificazione), tra classi ed interfacce staticamente nidificate

Classi Statiche: Esempio (1)

- Un utile esempio si trova nella classe `java.util.HashMap`
- La classe `HashMap` contiene al suo interno una tavola hash per la rappresentazione della mappa
- Quando si chiama il metodo `put()` la coppia passata al metodo (`<chiave>`, `<valore>`) deve essere memorizzata all'interno della tavola hash (per gestire le collisioni tra codici hash)

Classi Statiche: Esempio (2)

- La coppia viene memorizzata all'interno di un tipo nidificato statico `Entry<K,V>` in `HashMap<K,V>`
 - `Entry<K,V>` è anche il nome di una interfaccia nidificata nell'interfaccia `Map<K,V>`
 - `Map`, in quanto *capostipite* della gerarchia di mappe in Java, include anche una rappresentazione astratta delle coppie chiave-valore ivi contenute
- La *classe* `Entry<K,V>` in `HashMap<K,V>` implementa l'*interfaccia* `Entry<K,V>` in `Map<K,V>`

Classi Statiche: Esempio (3)

- E' possibile iterare su una mappa usando

```
Map<String, Integer> nome2eta = new HashMap<>();
```

```
...
```

```
for (Entry<String, Integer> e : nome2eta.entrySet())  
    System.out.println(e.getKey() + " " + e.getValue());
```

- Si sta facendo riferimento all'interfaccia **Entry** e non alla classe statica nidificata di **HashMap**
 - si itera allo stesso modo sulle coppie chiave-valore senza dover conoscere il tipo dinamico concreto della mappa in cui sono contenute

Classi Statiche: Esempio (4)

- La coppia viene memorizzata all'interno di una classe nidificata statica chiamata **Entry**

```
public class HashMap<K,V> ... {  
    Entry[] table; // tavola hash: un'array di  
        Entry  
  
    static class Entry<K,V> implements Map.Entry<K,V>{  
        final K key;  
        V value;  
        Entry<K,V> next; // lista di trabocco  
        ...  
    }  
}
```

Classi Statiche: Esempio (5)

- L'utilizzo di una classe statica nidificata per la classe **Entry** è ben motivata:
 - Una mappa e le coppie ivi contenute sono concetti ben distinti ma strettamente legati
 - Tuttavia la classe **Entry** non ha bisogno di un riferimento all'oggetto di tipo **Map** che la ospita
 - Ha quindi senso definirle assieme, all'interno della classe **HashMap<K,V>**, ma come semplice classe nidificata statica

Conclusioni

- Le classi nidificate (di varie tipologie) completano il quadro dei meccanismi offerti dal linguaggio Java per la definizione di classi (e di tipi, in generale)
 - Le lambda function di Java 8+ non introducono nuovi meccanismi di definizione di tipo
- Facilmente rimpiazzabili con le classi ordinarie, ma consentono una modellazione dei tipi più efficace per molti casi in cui si vuole precisare una relazione più stretta tra due tipi
- Interessante che il concetto di classe nidificate è totalmente sconosciuto alla JVM: il compilatore si occupa di “tradurre” le classi nidificate in classi ordinarie
- Fu forse il primo significativo passo di allontanamento della JVM dal linguaggio Java: la JVM ha poi intrapreso un percorso più indipendente dal linguaggio stesso
- Oggi la JVM viene utilizzata per molti altri linguaggi
 - Anche i più accaniti critici del linguaggio Java riconoscono l'enorme successo della JVM sottostante