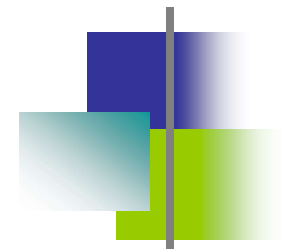




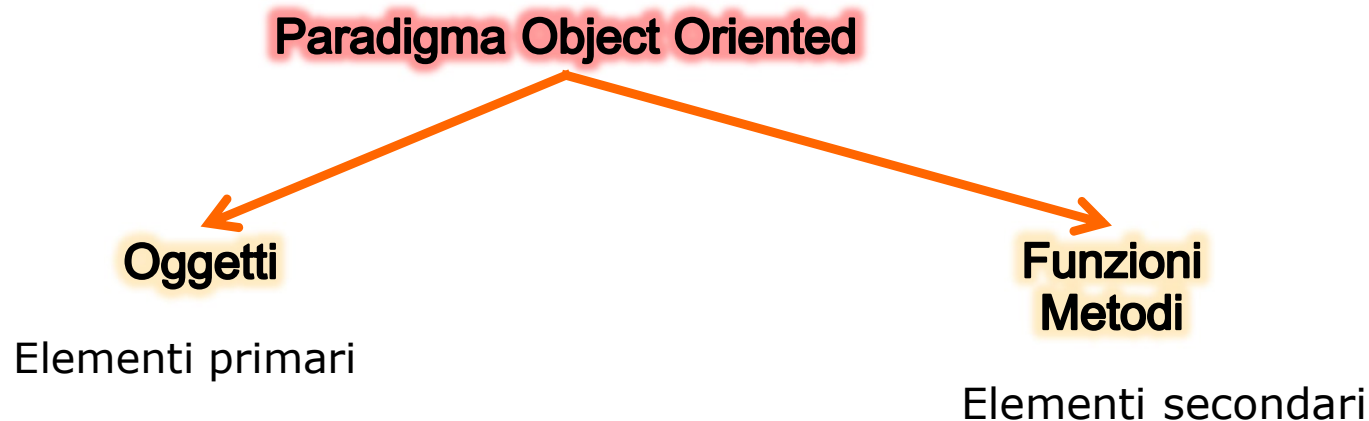
Interfacce funzionali e espressioni lambda

Dott. Romina Fiorenza
fiorenza.romina@gmail.com



Programmazione funzionale

- Java storicamente nasce (1995) come linguaggio Object Oriented, in contrapposizione con il paradigma dominante del tempo, cioè il paradigma funzionale.
- La programmazione ad oggetti pone al centro dell'attenzione **l'oggetto come dato condiviso e mutabile**.
 - Elementi come metodi o classi sono considerati **accessori necessari** alla manipolazione degli oggetti, cioè **elementi secondari**





Un esempio

- Supponiamo di voler scrivere un metodo che filtra su una collezione in base ad certo criterio: vogliamo filtrare mele in base al loro colore ed anche in base al loro peso. Si dovrebbero scrivere 2 metodi separati:

```
public static List<Mela> filtraMelePerColore(List<Mela> cassetta) {  
    List<Mela> listaFiltrata = new ArrayList<Mela>();  
    for(Mela m: cassetta) {  
        if(m.getColore().equals("verde")) listaFiltrata.add(m);  
    }  
    return listaFiltrata;  
}
```

```
public static List<Mela> filtraMelePerPeso(List<Mela> cassetta) {  
    List<Mela> listaFiltrata = new ArrayList<Mela>();  
    for(Mela m: cassetta) {  
        if(m.getPeso() > 150) listaFiltrata.add(m);  
    }  
    return listaFiltrata;  
}
```



Il predicato

- Come si può notare dall'esempio, i due metodi si differenziano soltanto dalla **condizione** che determina se una mela deve essere aggiunta nella lista
 - In matematica, una tale condizione (che ritorna un boolean) **si chiama predicato**
- Mentre in Java 7 è necessario avere entrambi i metodi, la nuova **espressione lambda di Java 8** consente di passare un predicato come parametro di un metodo

Si avrebbe così un solo metodo la cui firma sarebbe:

```
public static List<Mela> filtraMele(List<Mela> cassetta, Predicate<Mela> p) {  
    ...  
}
```

invocabile secondo una nuova forma, ad esempio questa, se voglio filtrare per peso:

```
filtraMele(cassetta, (Mela m) -> m.getPeso() > 150 );
```

oppure questa se voglio filtrare sul colore verde

```
filtraMele(cassetta, (Mela m) -> m.getColore().equals("verde") );
```

Metodi come parametri di metodi

- Il concetto chiave è il seguente: **una funzione può essere passata come parametro di un metodo**
 - Si parla quindi di programmazione **secondo uno stile funzionale**
 - Un metodo accetta come argomenti → primitivi, oggetti e funzioni
- Java 8 introduce una nuova sintassi per le **espressioni lambda** allo scopo di passare funzioni come parametri di metodi



I principali vantaggi sono i seguenti:

- migliore modularizzazione del codice
- codice sintetico e leggibile
- predisposizione al cambiamento dei requisiti

Elementi chiave

- Questa nuova caratteristica di java si basa su 3 elementi chiave:
 - Functional interface** → un'interfaccia che contiene un solo metodo astratto (per esempio potrebbe essere un predicato `Predicate<T>` di Oracle oppure altra interfaccia personalizzata)
 - Behavior parameterization** → metodo che accetta come argomento un comportamento. Cioè un metodo che prende come argomento un'interfaccia funzionale
 - Lambda expression** → nuova sintassi (per l'invocazione) che passa una funzione in alternativa all'oggetto che implementa l'interfaccia

1

```
public interface Predicate<T> {  
    boolean test(T m);  
}
```

JAVA 8:
Il metodo lavora con un **oggetto** variabile `obj` e un **comportamento** variabile dato dal predicato `p`

2

```
public static void esegui(String obj, Predicate<String> p) {  
    System.out.println(p.test(obj));  
}
```

dato

3

```
esegui("ciao", (String s) -> s.length() > 3);
```

comportamento

Otengo la stampa `true` perché la stringa `ciao` ha una lunghezza maggiore di 3



Espressioni lambda

- Le espressioni lambda sono:
 - **Funzioni** → sono funzioni perché il codice non si riferisce ad una particolare classe come accade ai normali metodi
 - **Anonime** → sono funzioni prive di nome
 - **Passabili** → si possono passare ad un metodo come normali argomenti
 - **Concise** → sintassi non verbosa al contrario di quello che accade con le inner class anonime
- **Nota:** le espressioni lambda non introducono nuove caratteristiche al linguaggio Java; si tratta più che altro di un nuovo stile molto più flessibile e pulito
- **Nota:** Java evita di introdurre il tipo funzione. In alternativa utilizza le interfacce funzionali.



Sintassi espressione lambda

- L'espressione lambda rappresenta il metodo astratto dell'interfaccia funzionale che intendiamo implementare
- La sintassi generale è :

```
( lista_parametri ) -> { codice_metodo }
```
- Conseguenza: verrà definita la classe col metodo astratto che segue le indicazioni specificate e infine ne verrà istanziato l'oggetto

Sintassi espressione lambda

- Primo caso: **espressione**

```
(Mela m1, Mela m2) -> m1.getColore().compareTo(m2.getColore());
```

return implicito

parametri freccia corpo della funzione (**espressione**)

- Primo caso: **istruzioni**

```
(Mela m1) -> {System.out.println(m1.getColore());}
```

Notare le parentesi graffe

parametri freccia corpo della funzione (**istruzioni**)

La differenza sta nel tipo di ritorno del metodo dell'interfaccia funzionale

- Corpo della funzione **espressione** → metodo ritorna qualcosa (non void)
- Corpo della funzione **istruzioni** → metodo ritorna void



Sintassi espressione lambda

- Una interfaccia Funzionale abilita all'invocazione stile lambda e deve seguire alcune regole sintattiche:
 - Deve essere una interfaccia
 - Può utilizzare i generics
 - Deve avere esattamente un solo metodo **astratto**
 - Il metodo può ritornare qualsiasi cosa, può avere qualsiasi parametro o sollevare qualsiasi eccezione
- Esempi

interfaccia	→	invocazione lambda
public interface A {boolean test(Mela m);}	}	(Mela m) → m.getColor().equals("red")
public interface B {boolean test();}	}	() → new Random().nextInt(100)>50
public interface C {void test(Mela m);}	}	(Mela m) → { System.out.println(m); } (Mela m) → System.out.println(m) <small>Singola riga</small>
public interface D {int test(String s);}	}	(String s) → s.length()
public interface E {int test(int a, int b);}	}	(int v1, int v2) → v1* v2



@FunctionalInterface

- Java 8 introduce una annotazione per marcare le interfacce funzionali: **@FunctionalInterface**
- L'annotazione è facoltativa e serve ad indicare che una certa interfaccia è da considerarsi una interfaccia funzionale
 - In questo caso, il compilatore controllerà le regole sintattiche per le interfacce funzionali come ad esempio la presenza di un unico metodo
- Dal punto di vista pratico, svolge lo stesso ruolo della annotazione **@Override**
 - Come **@Override**, è buona pratica utilizzarla



Le interfacce funzionali

- Attenzione alla regola:
 - Una interfaccia per essere funzionale deve avere esattamente metodo solo astratto.
 - Questa regola va rispettata anche in presenza di una gerarchia di interfacce
 - Una interfaccia (per essere) funzionale può estendere altre interfacce ma queste possono avere solo metodi default o static

```
@FunctionalInterface
interface InterfaceA {
    void metodo1();
}

@FunctionalInterface
interface InterfaceB extends InterfaceA{
    void metodo2();
}
```

InterfaceA è funzionale, mentre **InterfaceB** NON lo è.
L'annotazione darà errore.

NB: Se avessero lo stesso metodo, allora sarebbero entrambe funzionali.
InterfaceB maschererebbe il metodo di A, quindi possiederebbe comunque un solo metodo astratto

Sintassi espressione lambda: assegnazioni

- Secondo quanto appreso, anche la seguente assegnazione è valida:

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

Le parentesi bisogna scriverle
vuote perché il metodo run()
non prende parametri

```
//assegnazione lambda  
Runnable r = () → System.out.println("funziona!");  
Thread t = new Thread(r);  
t.start();
```

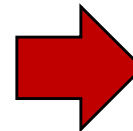
- Prima della versione 8, esistevano già diverse interfacce funzionali come ad esempio:
 - Runnable
 - Comparator<T>
 - Callable<V>

e molte altre...

Sintassi espressione lambda: type checked

- Le espressioni lambda sono **type checked**, cioè il compilatore è in grado di controllare la validità dei tipi all'interno delle espressioni.
- Ad esempio, supponiamo di avere l'interfaccia funzionale **MelaPredicate**, il cui metodo prende un oggetto **Mela** e ritorna un boolean:

```
public interface MelaPredicate {  
    boolean test(Mela m);  
}
```



Chiamata Lambda

Mela → **boolean**

```
MelaPredicate p = (Mela m) → m.getPeso() > 150;
```

Parametro Mela

Ritorno boolean

Sintassi espressione lambda: type inference

- E' possibile semplificare ancora un po' la sintassi sfruttando il **type inference** del compilatore.
 - Se il compilatore è in grado di determinare il tipo del parametro è possibile ometterlo nella espressione lambda.

```
public interface Predicate<T> {  
    boolean test(T m);  
}
```

```
public static void esegui(String obj, Predicate<String> p)  
{  
    System.out.println(p.test(obj));  
}
```

```
esegui("ciao", (String s) -> s.length() > 3);
```

```
esegui("ciao", s -> s.length() > 3);
```

EQUIVALENTI

Tipo dedotto in automatico dal compilatore.

- NOTA: se ci fossero 2 o più parametri, sono necessarie le parentesi tonde



Le interfacce funzionali di Java 8

- Java 8 fornisce le seguenti interfacce funzionali

Interfaccia	Chiamata lambda	Specializzazioni
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>



Le interfacce funzionali di Java 8

Interfaccia	Chiamata lambda	Specializzazioni
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiConsumer<L, R>	(L, R) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>
BiPredicate<T,U>	(T, U) → boolean	



Le interfacce funzionali di Java 8

- Notare che le interfacce funzionali offerte da Java 8 utilizzano i generics per aumentare le possibilità di utilizzo.
- L'uso dei generics però esclude i primitivi che possono essere utilizzati solo attraverso i wrapper Java (con o senza autoboxing)
 - L'autoboxing usa implicitamente i wrapper

Usare i wrapper esplicitamente o usare la tecnica del boxing ha un costo computazionale che potrebbe essere non trascurabile.

- Per evitare l'uso dei wrapper, Java offre interfaccia come specializzazione delle principali che contemplano l'uso diretto dei primitivi

Predicate<T>	T → boolean	IntPredicate, LongPredicate, DoublePredicate	int → boolean long → boolean double → boolean
---------------------------	--------------------	---	--



Interfacce funzionali nelle Collection

- Da java 8, sono disponibili nel framework delle Collection alcuni metodi di default (già concretamente implementati), che utilizzano le interfacce funzionali.
- I metodi sono così assegnati:
 - `Iterable` → `forEach(Consumer)`
 - `Collection` → `removeIf(Predicate)`
 - `List` →
 - `replaceAll(UnaryOperation)`
 - `sort(BiFunction)`
 - NB: fino a java 7 questi 2 metodi erano disponibili in `Collections`



Esempio d'uso

```
ArrayList<String> fiori = new ArrayList<String>();  
fiori.add("rosa");  
fiori.add("narciso");  
fiori.add("margherita");  
fiori.add("iris");  
fiori.removeIf(s -> s.endsWith("a"));
```

PREDICATE >> T -> boolean

La lista contiene narciso, iris

```
fiori.replaceAll(s -> s.toUpperCase());
```

UNARY_OPERATOR >> T -> T

La lista contiene NARCISO, IRIS

```
fiori.sort((s,t) -> s.compareTo(t));
```

BI_FUNCTION >> T,U -> R

La lista contiene IRIS, NARCISO

```
fiori.forEach( s -> System.out.println(s));
```

CONSUMER >> T -> VOID

Stampa → IRIS, NARCISO



Creare un'interfaccia funzionale

- Schema per ottenere un predicato:

```
public interface MyFunctional<T>{  
    public boolean myMethod(T type);  
}
```

- Schema per ottenere un consumer:

```
public interface MyFunctional<T>{  
    public void myMethod(T type);  
}
```

- Schema per ottenere un supplier:

```
public interface MyFunctional<T>{  
    public T myMethod();  
}
```



Creare un'interfaccia funzionale

- Schema per ottenere un unary operator:

```
public interface MyFunctional<T>{  
    public T myMethod(T type);  
}
```

- Schema per ottenere una function:

```
public interface MyFunctional<T, R>{  
    public R myMethod(T type);  
}
```

- Schema per ottenere una bi function:

```
public interface MyFunctional<T, U, R>{  
    public R myMethod(T type, U other);  
}
```