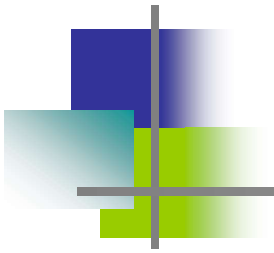
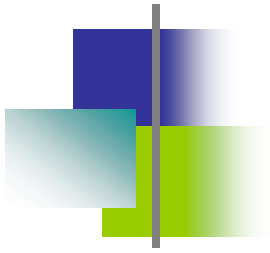


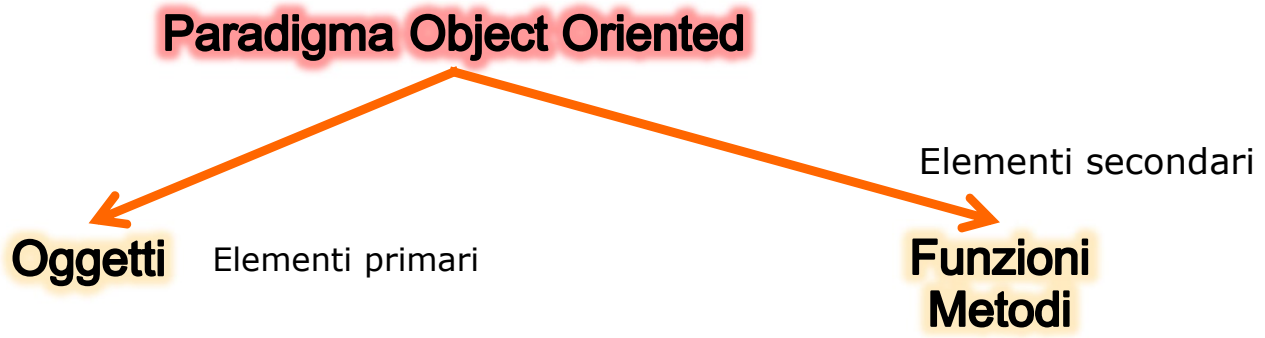
Interfacce funzionali e espressioni lambda



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Programmazione funzionale

- Java nasce storicamente 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**



L'idea è quella di trattare le funzioni come elementi primari, come se fossero oggetti

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Il concetto di predicato

- Supponiamo di voler tutti i file nascosti di una directory. Utilizzando i metodi della classe File si ottiene:

```
File[] hiddenFiles = new File(".").listFiles( new FileFilter() {
    public boolean accept (File file) {
        return file.isHidden();
    }
});
```

il codice (per quanto composto da sole 3 righe) è molto verboso se si considera di avere già metodi (ad alto livello) della classe File a disposizione

- Il problema è che in Java (7) i dati sono trattati come oggetti (elementi primari) e quindi anche **un predicato** deve essere trattato come oggetto
- **Java 8 introduce il concetto di funzione come parametro di un metodo**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Il concetto di predicato

- A ben vedere, il parametro del metodo listFiles è un pezzo di codice "camuffato" (wrappato) da oggetto Java

```
File[] hiddenFiles = new
File(".").listFiles(

    new FileFilter() {
      public boolean accept (File file){
          return file.isHidden();
      }
    }


);
```

funzione o predicato wrappato in un oggetto Java

Necessario in quanto in Java (7) i parametri di un metodo possono essere solo oggetti o primitivi

- In Java 8 la stessa operazione può essere fatta con il seguente codice:

```
File[] hiddenFiles = new File(".").listFiles((File f) -> f.isHidden());
```

comportamento passato come parametro di un metodo

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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**
  - Ad un metodo è possibile passare primitivi, oggetti e funzioni

- Java 8 introduzione 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

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Ancora 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;
}
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Ancora un esempio

- 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 cosi 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 );
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi

- Ritorniamo al problema delle mele:
  - Il concetto di parametrizzazione dei soli dati ci porterebbe ad avere:

```
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> filtraMelePerColore(List<Mela> cassetta, String colore) {
   List<Mela> listaFiltrata = new ArrayList<Mela>();
   for(Mela m: cassetta)
      if(m.getColore().equals(colore)) listaFiltrata.add(m);
   return listaFiltrata;
}
```

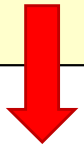
```
List<Mela> meleVerdi = filtraMelePerColore(cassetta, "verde");
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi

- Ritorniamo al problema delle mele:
  - Facciamo lo stesso per il peso

```
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;
}
```



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

```
List<Mela> melePesanti = filtraMelePerPeso(cassetta, 150);
```

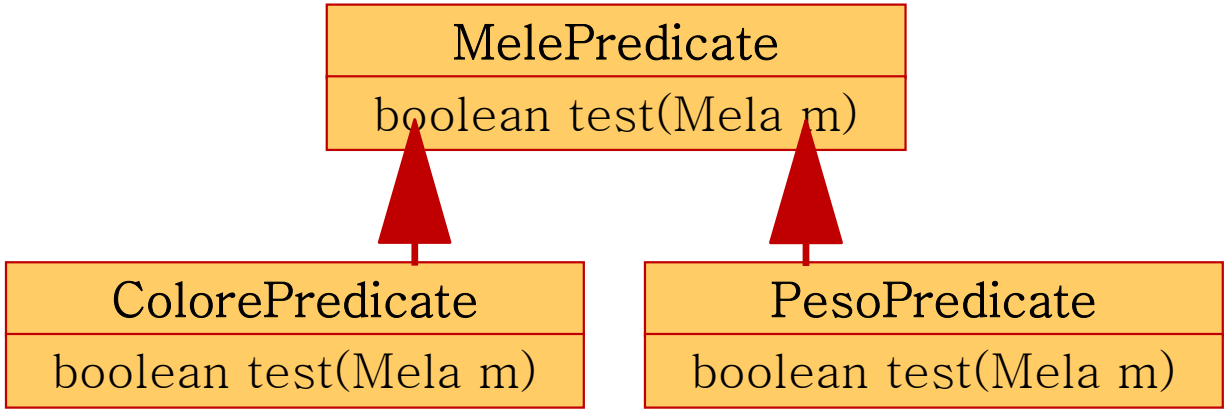
Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi: pattern Strategy

- Ma per parametrizzare il comportamento, cosa possiamo fare?
- Possiamo implementare il pattern strategy realizzando una interfaccia con una condizione testabile su un certo oggetto (quindi che ritorna un booleano)

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

- Si utilizza per applicare una diversa strategia di comportamento (pattern strategy appunto!)



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi: pattern Strategy

- Questo ci porta a scrivere un metodo generico che filtra sulle mele:

```
public static List<Mela> filtraMele(List<Mela> cassetta, MelePredicate p) {
    List<Mela> listaFiltrata = new ArrayList<Mela>();
    for(Mela m: cassetta)
      if(p.test(m)) listaFiltrata.add(m);
   return listaFiltrata;
}
```

a patto di avere una o più classi di tipo MelePredicate (le diverse strategie)

```
public class ColorePredicate implemente MelePredicate {
   public boolean test(Mela m) { return m.getColore().equals("verde"); }
}
```

```
public class PesoPredicate implemente MelePredicate {
   public boolean test(Mela m) {return m.getPeso() > 150; }
}
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi: inner class

- Per diminuire la verbosità, java introduce sin da subito le inner class anonime: quindiclassi di tipo Predicato

Invocazione esplicita:

```
MelePredicate p = new ColorePredicate();
List<Mela> listaFiltrata = filtraMele(cassetta, p);
```

Invocazione via inner class anonima:

```
List<Mela> listaFiltrata = filtraMele(cassetta, new MelePredicate(){
   public boolean test(Mela m) {
       return m.getColore().equals("verde");
   }
});
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi: espressioni lambda

- Java 8 attraverso le espressioni lambda ingegnerizza e riorganizza questo scenario

Invocazione esplicita (Java 7):

```
MelePredicate p = new ColorePredicate();
List<Mela> listaFiltrata = filtraMele(cassetta, p);
```

Invocazione via inner class anonima (Java 7):

```
List<Mela> listaFiltrata = filtraMele(cassetta, new MelePredicate(){
   public boolean test(Mela m) {
       return m.getColore().equals("blu");
   }
});
```

Invocazione via espressione lambda (Java 8):

Nuovo operatore ->

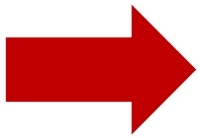
```
List<Mela> lista = filtraMele(cassetta, (Mela m) -> m.getColore().equals("blu"));
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Procediamo per gradi: generics

- Per migliorare ulteriormente lo schema si potrebbero utilizzare i generics. L'interfaccia MelePredicate, si potrebbe definire come generica interfaccia che rappresenta un predicato per poi parametrizzarla via generics

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



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

Java 8

- Java 8 introduce tale intefaccia nel package **java.util.function**.

- Anche la funzione di filtro si può generalizzare attraverso i generics

```
public static <T> List<T> filtra(List<T> lista, Predicate<T> p) {
    List<T> listaFiltrata = new ArrayList<>();
    for(T t: lista)
      if(p.test(t)) listaFiltrata.add(t);
   return listaFiltrata;
}
```

Utilizzo dei generics a livello di metodo

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Ricapitoliamo

- Per sfruttare a pieno le nuove possibilità abbiamo bisogno di 3 elementi:
  1. **Functional interface** → l'interfaccia che rappresenta un predicato (come Predicate<T>)
  2. **Behavior parameterization** → metodo che ha come argomento un comportamento. Cioè un metodo che prende come argomento un predicato
  3. **Lambda expression** → nuova sintassi per passare una funzione in sostituzione dell'oggetto predicato

1
```
public interface Predicate<T> {
    boolean test(T m);
}
```

2
```
public static void esegui(String obj, Predicate<String> p) {
    System.out.println(p.test(obj));
}
```

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

JAVA 8:
Diverso comportamento (behavior)
Con diversi dati (object)

dato

comportamento

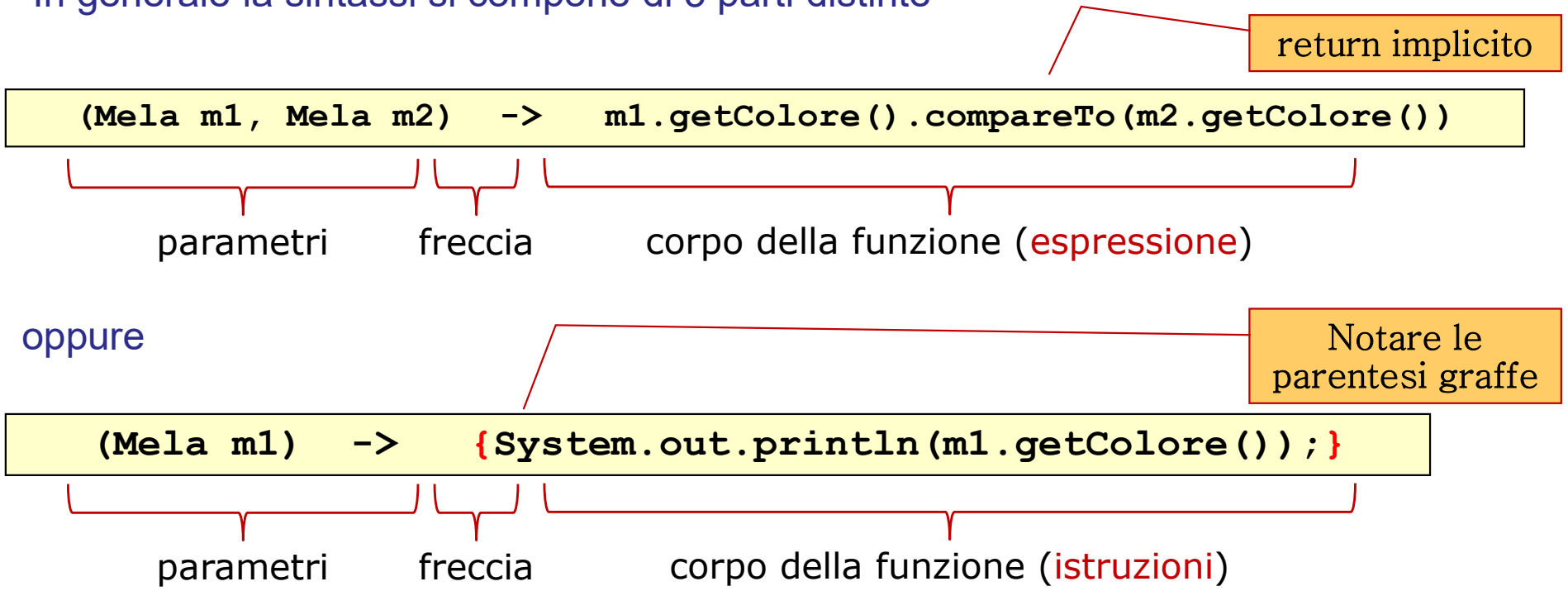
Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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. Piuttosto utilizza le interfacce funzionali.

Sintassi espressione lambda

- In generale la sintassi si compone di 3 parti distinte



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

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

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Sintassi espressione lambda

- Una interfaccia Funzionale (come Predicate) regolamenta l'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

`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)` Singola riga
`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`

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Le interfacce funzionali

- Attenzione alla regola:
  - Una interfaccia per essere funzionale deve avere esattamente metodo solo astratto.
  - Questo 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
public interface InterfacciaA {
      void esegui();
}

@FunctionalInterface
interface InterfacciaB extends InterfacciaA{
      void esegui();
}
```

Sono entrambe interfacce funzionali compreso B.
L'interfaccia B pur estendendo A non eredita ma maschera il metodo esegui di A.
Quindi possiede un solo metodo

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Sintassi espressione lambda: assegnazioni

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

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

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

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

- Java, anche prima della versione 8, offriva già diverse interfacce funzionali come ad esempio:
  - Runnable
  - Comparator<T>
  - Callable<V>

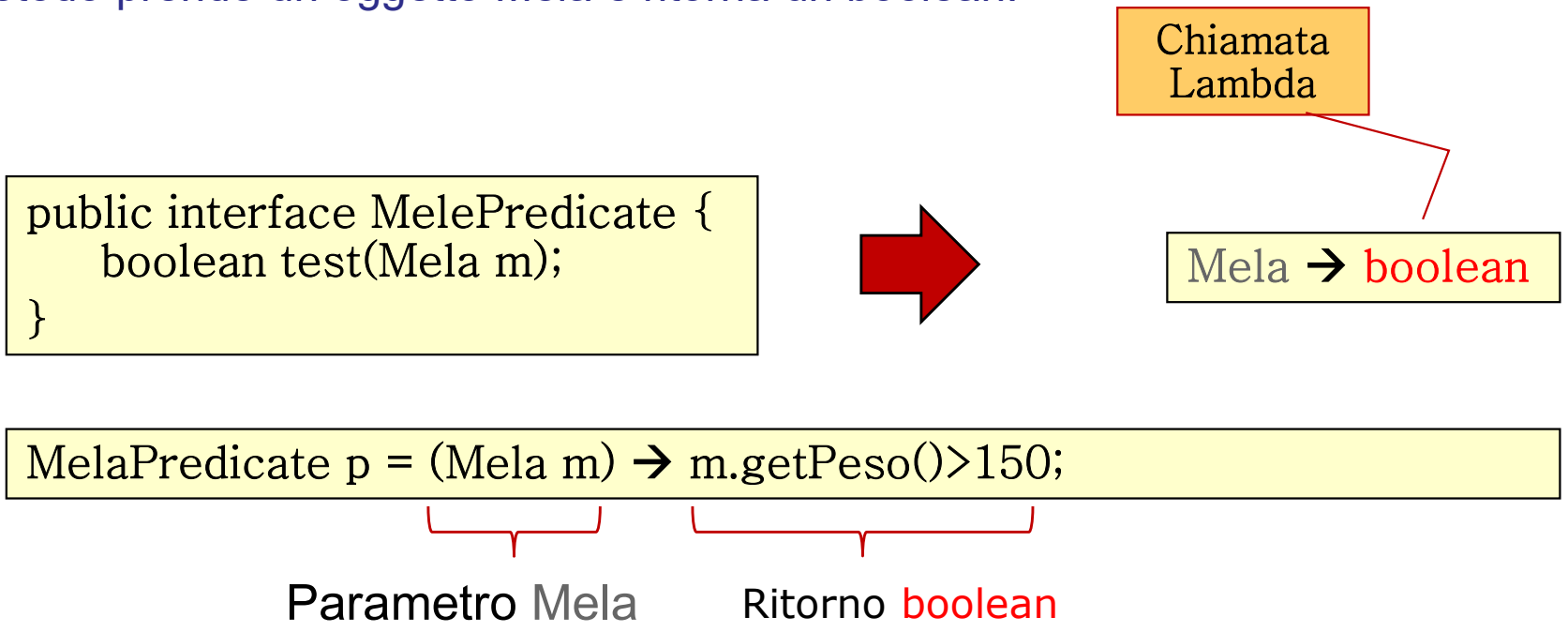
  e molte altre…

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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:



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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));
}
```

EQUIVALENTI

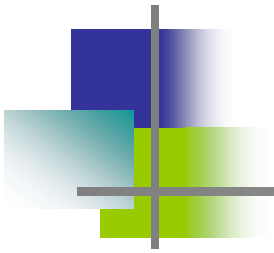
```
esegui("ciao", (String s) -> s.length() >3);
```

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

Tipo dedotto in automatico dal compilatore.

- NOTA: se i parametri fossero 2 o più, sarebbe necessario aggiungere le parentesi tonde

Dott. Romina Fiorenza
fiorenza.romina@gmail.com



@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

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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>

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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	

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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) -> t.compareTo(s));
```

BI_FUNCTION >> `T,U -> R`

La lista contiene **IRIS, NARCISO**

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

CONSUMER >> `T -> VOID`

Stampa → **IRIS, NARCISO**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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();
}
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

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);
}
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com