

Java

- I nuovi paradigmi e linguaggi tendono a semplificare il lavoro del programmatore, nascondendo dentro le librerie (o nei costrutti del linguaggio) parte della programmazione precedentemente necessaria
- Da Java 5 (settembre 2004), in un ciclo, si può scorrere una lista, o un array, senza dover dare il valore di inizio e fine dell'indice (*ciclo for avanzato*)

```
List<String> nomi;  
...  
for (String nome : nomi) ...
```
- Da Java 7 (luglio 2011), si può evitare di indicare il tipo di elementi nella collection al momento dell'istanziatura, lasciando al compilatore l'inferenza, ovvero

```
List<String> nomi = new ArrayList<>();
```
- Java è un linguaggio imperativo, tuttavia Java 8 (marzo 2014) include caratteristiche tipiche della programmazione funzionale. La programmazione funzionale è in genere più concisa, espressiva, e facile da parallelizzare rispetto alla programmazione ad oggetti

Prof. Tramontana - Maggio 2020

Espressioni Lambda

- Il codice sotto è un'espressione lambda
`s -> System.out.println("Ciao, " + s)`
- Un'espressione lambda è una funzione anonima, che prende in ingresso parametri, con nome dato a sinistra del segno freccia, e un blocco di codice, a destra del segno freccia. E' simile a un metodo: ha una lista di parametri, un corpo e un tipo di ritorno. Può essere passata come argomento di un metodo. E' concisa: non serve scrivere preamboli
- Nei linguaggi funzionali puri, un'espressione lambda è una funzione pura, ovvero il risultato dipende solo dagli input (non ha uno stato). In Java, un'espressione lambda può accedere a uno stato esterno (può non essere pura)
- La seguente è un'espressione lambda che prende in ingresso due parametri, `x` e `y`, e implementa la somma, il risultato è il valore di ritorno
`(x, y) -> x + y`
- I parametri in ingresso sono `x` e `y`, a sinistra della freccia, il cui tipo è determinato automaticamente. Quando non ha parametri, o ne ha più di uno, occorre racchiudere i parametri fra parentesi tonde
`() -> System.out.println("Buongiorno")`
- Il corpo della funzione anonima è il codice a destra della freccia, nel caso di più di un'istruzione, occorre racchiuderle fra parentesi graffe
`(x, y) -> { System.out.println("x: " + x); return x+y; }`

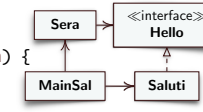
3

Prof. Tramontana - Maggio 2020

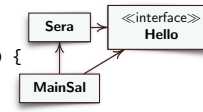
Classi Anonime In Java

- Da Java 1.1 si possono usare classi anonime per implementare interfacce

```
public interface Hello {  
    public void greetings(String s);  
}  
  
public class Sera {  
    private Hello myh;  
    public Sera(Hello h) {  
        myh = h;  
    }  
    public void saluti() {  
        myh.greetings("buonasera");  
    }  
}  
  
public class Saluti implements Hello {  
    public void greetings(String s) {  
        System.out.println("Ciao, " + s);  
    }  
}  
  
public class MainSal {  
    public static void main(String[] args) {  
        Sera sr = new Sera(new Saluti());  
        sr.saluti();  
    }  
}
```



```
public interface Hello {  
    public void greetings(String s);  
}  
  
public class Sera {  
    private Hello myh;  
    public Sera(Hello h) {  
        myh = h;  
    }  
    public void saluti() {  
        myh.greetings("buonasera");  
    }  
}  
  
public class MainSal {  
    public static void main(String[] args) {  
        Sera sr = new Sera(new Hello() {  
            public void greetings(String s) {  
                System.out.println("Ciao, " + s);  
            }  
        });  
        sr.saluti();  
    }  
}
```



2

Prof. Tramontana - Maggio 2020

Implementazione Interfaccia

```
public interface Hello {  
    public void greetings(String s);  
}
```

- La classe anonima implementa l'interfaccia Hello con codice fornito nel punto dove si istanzia

```
Sera sr = new Sera(new Hello() {  
    public void greetings(String s) {  
        System.out.println("Ciao, " + s);  
    }  
});  
sr.saluti();
```

- Con Java 8 e le espressioni lambda si evitano i preamboli (o codice standard, boilerplate), e si implementa solo il comportamento, quando l'interfaccia ha un singolo metodo

```
// Java 8  
Sera sr = new Sera(s2 -> System.out.println("Ciao, " + s2));  
sr.saluti();
```

- L'espressione lambda implementa il metodo dichiarato in Hello, e si passa al costruttore di Sera. Per il compilatore, `s2` è di tipo String poiché così è il parametro del metodo in Hello

4

Prof. Tramontana - Maggio 2020

Cerca Valori Su Lista

```
public class Trova {
    private List<String> listaNomi = Arrays.asList("Nobita", "Nobi",
        "Suneo", "Honekawa", "Shizuka", "Minamoto", "Takeshi", "Gouda");

    // in stile imperativo
    public void trovaImper() {
        boolean trovato = false;
        for (String nome : listaNomi)
            if (nome.equals("Nobi")) {
                trovato = true;
                break;
            }
        if (trovato) System.out.println("Nobi trovato");
        else System.out.println("Nobi non trovato");
    }

    // in stile dichiarativo
    public void trovaDichiar() {
        if (listaNomi.contains("Nobi")) System.out.println("Nobi trovato");
        else System.out.println("Nobi non trovato");
    }
}
```

5

Prof. Tramontana - Maggio 2020

Stile Funzionale

- Lo stile di programmazione funzionale è dichiarativo. La programmazione funzionale aggiunge allo stile dichiarativo **funzioni di ordine più alto**, ovvero funzioni che hanno come parametri altre funzioni
- In Java si possono passare oggetti ai metodi, creare oggetti dentro i metodi, e ritornare oggetti dai metodi
- In Java 8 si possono passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi**
- Un metodo è una parte di una classe, mentre una funzione non è associata ad una classe
- Un metodo o una funzione che ricevono, creano o ritornano una funzione, si considerano essere **funzioni di ordine più alto**

7

Prof. Tramontana - Maggio 2020

Considerazioni

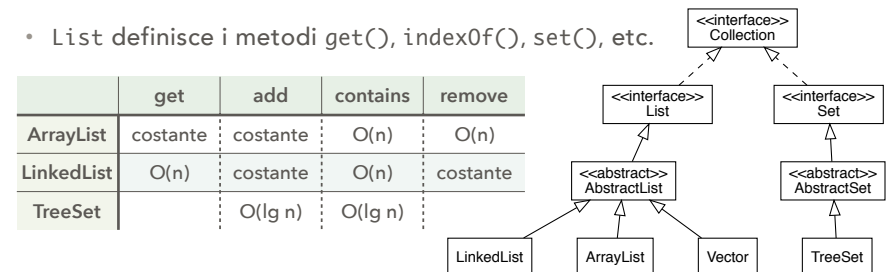
- L'implementazione **imperativa** si serve di una variabile boolean come flag per indicare se l'elemento è stato trovato. Inoltre si usa un ciclo per scorrere la lista e controllare se uscire dal ciclo
- Lo stile imperativo dà al programmatore il controllo di quel che il programma deve fare, tuttavia
 - Bisogna implementare varie linee di codice, e molto spesso si hanno cicli che scorrono liste per trovare valori, calcolare somme, etc.
 - Per capire cosa si vuol fare, dobbiamo prima leggere tanti dettagli all'interno del corpo del ciclo
 - Il ciclo è **esterno** al codice che implementa la lista (l'iterazione è **esterna**)
- Nella versione **dichiarativa** vari dettagli dell'implementazione sono nella libreria sottostante (sul metodo contains() della classe ArrayList), l'iterazione è **interna**

6

Prof. Tramontana - Maggio 2020

Java Collection

- Le librerie di Java hanno tante interfacce e classi collection. Collection è un'interfaccia che definisce i metodi add(), remove(), size(), contains(), containsAll(), etc.
- List definisce i metodi get(), indexOf(), set(), etc.



- ArrayList è un array espandibile: cresce del 50% quando non vi è spazio. **Gli elementi sono contigui, quindi l'accesso al generico elemento è veloce**
- LinkedList è una lista, ogni elemento ha i riferimenti al successivo e al prec
- Vector è simile ad ArrayList, ma è synchronized

8

Prof. Tramontana - Maggio 2020

Metodi Di Default

- Java 8 introduce i metodi di default per le interfacce
- Ciò ha permesso di includere nuovi metodi su interfacce già esistenti, senza compromettere la compatibilità delle applicazioni conformi a precedenti versioni di Java
- `stream()` è un metodo di default dell'interfaccia `Collection`
- I metodi di default non agiscono sullo stato, invece le classi astratte che hanno metodi implementati possono avere costruttori e metodi che agiscono sullo stato
- `stream()` restituisce uno `Stream<T>` che è una sequenza di elementi di tipo `T` (`T` è definito dalla `collection` su cui è invocato `stream()`). `stream()` permette di fare operazioni sugli elementi presenti sull'istanza di `collection` su cui è invocato
- Il tipo `Stream` definisce vari metodi che prendono in ingresso funzioni

9

Prof. Tramontana - Maggio 2020

Esempi Con Filter

- Contare quanti elementi della lista `nomi` hanno lunghezza 5 caratteri
 - Si noti che non si può usare il metodo `contains()` di `List` (che è invece utile per verificare se una lista contiene un certo elemento)

```
long c = nomi.stream()
    .filter(s -> s.length() == 5)
    .count();
```

- L'espressione `lambda s -> s.length() == 5` ha in ingresso `s`, ovvero un elemento dello stream; su `s` si invoca `length()` (metodo di `String` che restituisce la lunghezza di `s`), quindi si valuta se è pari a 5

- Contare quanti elementi della lista `nomi` sono stringhe vuote

```
long c = nomi.stream()
    .filter(s -> s.isEmpty())
    .count();
```

- `isEmpty()` è un metodo di `String` (non ha parametri di ingresso e restituisce un boolean). Tale metodo è passato a `filter()`, e sarà chiamato su ciascun elemento dello stream, quindi ciascun elemento dello stream è valutato da `isEmpty()`

11

Prof. Tramontana - Maggio 2020

Metodo Filter

- Si abbia una lista di `String`, contare quante volte è presente un certo valore

```
List<String> nomi = List.of("Nobita", "Nobi", "Suneo");
long c = nomi.stream()
    .filter(s -> s.equals("Nobi"))
    .count();
```

- `of()` è un metodo statico di `List` (da Java 9) che restituisce una lista **non-modificabile** contenente gli elementi passati
- `filter(Predicate<T> p)` è un metodo di `Stream`, **prende come argomento una funzione che ritorna un boolean** (quindi solo `true` o `false`). Una funzione che ritorna un boolean è detta **predicato**. `filter()` ritorna uno `Stream` costituito da tutti gli elementi della lista che soddisfano il predicato passato in input
- L'espressione `lambda s -> s.equals("Nobi")` è un predicato che ha in ingresso `s`, che è un elemento dello stream, e restituisce un boolean. Inoltre, `equals()` è un metodo di `String` che restituisce un boolean
- `filter()` è *lazy*, ovvero operazione intermedia, può essere eseguita in un momento successivo rispetto a dove è inserita, è eseguita quando è necessario
- `count()` è un metodo di `Stream`, è *eager*, ovvero operazione terminale, genera un valore e forza l'esecuzione delle precedenti operazioni
- Tutte le operazioni che restituiscono uno `Stream` sono operazioni intermedie

10

Prof. Tramontana - Maggio 2020

Tipo Predicate

- Si può definire una funzione che restituisce un boolean

```
Predicate<Integer> positive = x -> x >= 0;
```

- `Predicate` rappresenta un'interfaccia funzionale, ovvero un'interfaccia che definisce un solo metodo
- L'annotazione `@FunctionalInterface` indica al compilatore che l'interfaccia ha solo un metodo astratto, così il compilatore controlla che abbia un solo metodo (evita che versioni successive inseriscano altri metodi)
- `Predicate` definisce il metodo `test()` che prende in ingresso un parametro di tipo `Object`
- In questo esempio, il metodo `test()` ha parametro in ingresso `x` di tipo `Integer`, e la sua implementazione è `x >= 0`;
- Il predicato `positive` si può passare ad un metodo

```
Stream<Integer> result = Stream.of(2, 5, 10, -1)
    .filter(positive);
```

- Il metodo `of()` è un metodo static di `Stream` che costruisce uno `Stream`

12

Prof. Tramontana - Maggio 2020

Metodo Reduce

- `reduce(T identity, BinaryOperator<T> accumulator)` è un metodo di `Stream`, prende un valore dello stesso tipo degli elementi dello stream, e un'espressione lambda che ha due valori in ingresso e ritorna un valore
- `reduce()` si usa quando si vuol passare da un insieme di valori ad un singolo valore, per esempio per ottenere la somma

`reduce(0, (accum, v) -> accum + v);`

- `reduce()` applica la funzione di accumulazione che gli viene passata (secondo argomento), in questo caso la somma, a tutti gli elementi dello stream e ritorna un risultato dello stesso tipo dei valori nello stream
- Ciascun valore dello stream è rappresentato dal parametro `v`
- La somma calcolata per ciascun `v` è tenuta dalla variabile `accum` e inizialmente vale `0` (valore fornito dal primo parametro di `reduce()`)
- `reduce()` è un'operazione terminale

13

Prof. Tramontana - Maggio 2020

Reduce

```
public class Pagamenti {
    private List<Float> importi = new ArrayList<>();

    // in stile imperativo
    public float calcolaSommaImper() {
        float risultato = 0f;
        for (float v : importi)
            risultato += v;
        return risultato;
    }

    // in stile funzionale
    public float calcolaSomma() {
        return importi.stream()
            .reduce(0f, Float::sum);
    }
}
```

15

Prof. Tramontana - Maggio 2020

Riferimenti A Metodi

- La sintassi `::` permette di avere il riferimento a un metodo, che si può quindi passare a un altro metodo
- Es. `Integer::sum` è il riferimento al metodo statico `sum()` di `Integer` che prende due parametri `Integer` e ne ritorna la somma
- Passando il riferimento `Integer::sum` alla `reduce()`, verrà calcolata la somma degli elementi dello stream, come quando si passa l'espressione lambda `(s, v) -> s + v`

`reduce(0, Integer::sum)`

- `reduce()` può prendere in ingresso solo un'espressione lambda, e in tal caso restituisce un tipo `Optional<T>` che può contenere il risultato
- `reduce(Integer::sum)`
- Il risultato non esiste se lo stream di partenza è vuoto
- `Optional` rappresenta un valore che può esistere o meno, se esiste il metodo `isPresent()` darà un valore `true`; per estrarre il valore si usa `get()`

14

Prof. Tramontana - Maggio 2020

Metodo Map

- `map(Function<T, R> mapper)` di `Stream` prende in ingresso una funzione `mapper`, e restituisce uno stream contenente i risultati dell'esecuzione della funzione su ciascun elemento dello stream iniziale
- Ovvero, `map()` chiama su ciascun elemento dello stream iniziale la funzione passata e dà in uscita il risultato della funzione. Ciascun risultato è inserito in un nuovo stream
- `map()` è un'operazione intermedia

`map(x -> x * 2)`

- Eseguito su uno stream contenente valori interi, restituisce uno stream contenente i valori iniziali raddoppiati

`map(Persona::getEta)`

- Eseguito su uno stream di istanze di `Persona`, restituisce uno stream contenente i valori in uscita da `getEta()`, quest'ultimo è un metodo di `Persona` ed è invocato su ciascun elemento dello stream iniziale

16

Prof. Tramontana - Maggio 2020

```

public class Persona {
    private String nome;
    private int eta;
    public Persona(String n, int e) {
        nome = n;
        eta = e;
    }
    public String getNome() {
        return nome;
    }
    public int getEta() {
        return eta;
    }
}

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));

```

- Calcoliamo la somma delle età in stile funzionale

```

int somma = p.stream()
    .map(Persona::getEta)
    .reduce(0, Integer::sum);

```

```

// in stile imperativo
int somma = 0;
for (Persona x : p)
    somma += x.getEta();

```

- La funzione passata a map() è il metodo getEta() di Persona

17

Prof. Tramontana - Maggio 2020

Ricerca Su Lista (Imperativa)

- Data la lista di istanze di Persona, trovare il nome della persona che è più grande (di età) fra quelli che hanno meno di 20 anni

```

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));

```

- In versione imperativa, se volessimo scorrere la lista solo una volta

```

Persona pmax = null;
for (Persona x : p)
    if (x.getEta() < 20) {
        if (pmax == null) pmax = x;
        if (pmax.getEta() < x.getEta()) pmax = x;
    }
if (pmax != null) System.out.println("persona: " + pmax.getNome());

```

- Il corpo del ciclo ha varie condizioni, queste rendono il codice più difficile da comprendere

19

Prof. Tramontana - Maggio 2020

Dichiarativo—Funzionale

- Data una lista contenente valori String

```

List<String> nomi = Arrays.asList("Saro", "Taro", "Ian", "Al");

```

- Per determinare se la lista contiene un certo valore, in stile dichiarativo

```

if (nomi.contains("Saro")) System.out.println("Saro trovato");

```

- Data una lista contenente istanze di Persona

```

List<Persona> p = Arrays.asList(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));

```

- Lo stile funzionale consente di estrarre il campo nome, e inoltre permette di valutare una funzione ad-hoc, quindi è molto più flessibile e potente

```

long c = p.stream()
    .filter(s -> s.getNome().equals("Saro"))
    .count();

```

- count() conta quanti elementi ha lo stream prodotto da filter

18

Prof. Tramontana - Maggio 2020

Ricerca Funzionale, Versione 1

- Aggiungendo su Persona il metodo getMax()

```

/** restituisce l'istanza con il valore massimo di eta' */
public static Persona getMax(Persona p1, Persona p2) {
    if (p1.getEta() > p2.getEta())
        return p1;
    return p2;
}

```

- Possiamo implementare la ricerca in modo funzionale

```

Optional<Persona> pmax = p.stream()
    .filter(x -> x.getEta() < 20)
    .reduce(Persona::getMax);

if (pmax.isPresent())
    System.out.println("persona: " + pmax.get().getNome());

```

- filter() è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- reduce() è usata per selezionare un elemento: invoca getMax() che confronta a due a due

20

Prof. Tramontana - Maggio 2020

Ricerca Funzionale, Versione 2

```
Optional<Persona> pmax = p.stream()
    .filter(x -> x.getEta() < 20)
    .max(Comparator.comparing(Persona::getEta));

if (pmax.isPresent())
    System.out.println("persona: " + pmax.get().getNome());
```

- `filter()` è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- `max()` trova il valore massimo, è un'operazione terminale, prende un `Comparator`, restituisce un `Optional`, `max()` opera in modo simile a `reduce()`
- `comparing()` di `Comparator` prende una funzione che estrae una chiave e restituisce un `Comparator`
- `Comparator` implementa una funzione di confronto (`compare()`) che controlla l'ordinamento di una collezione di oggetti
- `max()` al suo interno chiama il metodo `compare()` del `Comparator` in input

21

Prof. Tramontana - Maggio 2020

Programmazione Parallela

- I processori attuali hanno vari core, tipicamente due per i portatili, quattro per i desktop, dodici per i server. La programmazione parallela è più difficile di quella sequenziale e usare bene l'hardware risulta più complicato
- In Java, la classe `Thread` permette di lanciare un nuovo thread di esecuzione, ma spesso bisogna risolvere i problemi di corsa critica, usando opportunamente `synchronized`, `wait`, `notify`
 - Java 5 mette a disposizione `Lock`, `Esecutori`, etc.
- Le operazioni `map()` e `filter()` di `Stream` sono stateless, ovvero non tengono uno stato durante l'esecuzione, quindi il risultato non dipende dall'ordine in cui i singoli elementi su cui eseguono vengono prelevati
- **Lo stream di origine non viene modificato e le operazioni `map()`, `filter()`, etc., generano uno stream distinto. Questo facilita grandemente la parallelizzazione**
- Attenzione: se l'applicazione durante l'esecuzione di un'operazione, per es. di `map()`, aggiorna uno stato globale, l'operazione non è più stateless

23

Prof. Tramontana - Maggio 2020

Metodo Collect

```
List<Persona> p = List.of(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- Ricaviamo la lista delle età
`List<Integer> e = p.stream()
 .map(x -> x.getEta())
 .collect(Collectors.toList());`
- Come prima, `map()` restituisce uno stream con i valori delle età, e la funzione passata dice come trasformare ciascun elemento dello stream
- `collect()` permette di raggruppare i risultati e prende in ingresso un `Collector`
- La classe `Collectors` implementa metodi utili per raggruppamenti, il metodo `toList()` restituisce un `Collector` che accumula elementi in una `List`

```
// in stile imperativo
List<Persona> p; // come sopra
List<Integer> e = new ArrayList<>();
for (Persona x : p)
    e.add(x.getEta());
```

22

Prof. Tramontana - Maggio 2020

Stream Paralleli

- `Collection` ha i metodi di default `stream()` e `parallelStream()`
- Il metodo `parallelStream()` possibilmente dà uno stream parallelo. Quindi si può avere l'esecuzione parallela basata su `parallelStream()` (niente più bisogno di thread e sincronizzazione, per molti casi)
- Le prestazioni dipendono da: (i) numero di elementi dello stream, (ii) operazioni da svolgere, (iii) hardware, e (iv) tipo di `Collection` su cui si invoca l'operazione
- La `Collection` potrebbe essere `ArrayList`, `LinkedList`, etc., con tempi di accesso diversi (vedere tabella precedente)
- Eseguendo su milioni di elementi il seguente codice, le prestazioni migliorano (su hardware con più core) per `ArrayList`, `Vector`, `TreeSet`; migliorano poco (o non migliorano) quando si usa `LinkedList` a causa dell'accesso sequenziale

```
long c = nomi.parallelStream()
    .map(String::toUpperCase)
    .filter(s -> s.equals("NOBI"))
    .count();
```

24

Prof. Tramontana - Maggio 2020

Hardware 4 core, Ricerca su 5 Milioni di elementi, Java 11.0.2

