Java 8 e Lambda Expressions

- **Perche**' e' stata aggiunta la programmazione funzionale in Java?
  - per **rinnovare il framework delle collezioni** in modo da:
    - supportare le ***parallel bulk operations***: operazioni di massa sui dati di una collezione di grandi dimensioni, da effettuare in parallelo con diversi thread
    - esprimere le operazioni sui dati come "funzionalità", indicando cioè ***"cosa"*** fare su ogni elemento invece di ***"come"*** farlo

lazy evaluation, out-of-order execution, In parallel…

Java 8 e Lambda Expressions

- Perche' e' stata aggiunta la programmazione funzionale in Java?

iterazione esterna

```
Collection<Person> people = …
int maxAge=-1;
for(Person p : people)
  if(p.getGender()==MALE && p.getAge()>maxAge)
    maxAge=p.getAge();
```

- No mutable **maxAge**
- Nicely composed
- Easier to enhance or change the logic
- Iteration controlled by library methods
- Efficient: lazy evaluation loops
- Easy to parallelize (no shared **maxAge**)

iterazione interna

```
Collection<Person> people = ...
int maxAge= people.stream()
                  .filter(p -> p.getGender()==MALE)
                  .mapToInt(p -> p.getAge())
                  .max();
```


Java 8 e Lambda Expressions

- Perche' e' stata aggiunta la programmazione funzionale in Java?
- Come e' stata aggiunta?
  - functional interfaces, lambda expressions, method references, enhanced type inference, default methods, stream
  - **arricchire le collezioni mantenendo la retrocompatibilita'!!**

Java 8 e Lambda Expressions

- come rappresentare le funzioni?
- come rappresentare il ***tipo*** funzione?
  - e.g. `int => double` significherebbe aggiungere complessita' al type system, e si puo' fare senza
  - usando interfacce con un solo metodo

```
interface Fun { double apply(int i); }
```

il subtyping per il tipo funzione? indirettamente usando generics…

Functional Interfaces

in `java.util.function`:

```
public interface Function<T,R> { R apply(T t); }
public interface Predicate<T> { boolean test(T t); }
public interface Supplier<T> { T get(); }
public interface Consumer<T> { void accept(T t); }
…
```

e le vecchie interfacce di callback:

```
public interface Runnable { public void run(); }
public interface ActionListener extends … {
                void actionPerformed(ActionEvent e);}
```

Lambda Expression

Int => Int

```
ToIntFunction<Integer> f1 = z -> z*10;
```

(Int,Int) => Int

```
ToIntBiFunction<Integer,Integer> f2 = (x,y) -> x-y;
```

(String,String) => Boolean

```
BiPredicate<String,String> f3 = (s1,s2) -> s1.equals(s2);
```

Int => Unit

```
Consumer<Integer> f4 = (Integer j) -> System.out.print(j);
```

Double => Double

```
Function<Double,Double> f5 = (Double d) ->
                                { if (d<10) return d-10;
                                  else return d; }
```


Lambda Expression

```
ToIntFunction<Integer> f1 = z -> z*10;
int x1 = f1.applyAsInt(18);

ToIntBiFunction<Integer,Integer> f2 = (x,y) -> x-y;
int x2 = f2.applyAsInt(20,4);

BiPredicate<String,String> f3 = (s1,s2) -> s1.equals(s2);
boolean x3 = f3.test("pippo","pluto");

Consumer<Integer> f4 = (Integer j) -> System.out.print(j);
f4.accept(11);

Function<Double,Double> f5 = (Double d) ->
                                { if (d<10) return d-10;
                                  else return d; }
Double x5 = f5.apply(3.5);
```


Lambda Expression

- non **`T => R`** ma **`Function<T,R>`**
- ogni tipo funzione ha un **nome di tipo** diverso
  - ToIntFunction vs Function<Integer,Integer>
  - Supplier<T> e Consumer<T> sono Unit =>T e T =>Unit
  - **facilita il type inference** avere nomi di tipo diversi invece di diversi parametri di tipo (…raw type dei Generics…)
- ogni functional interface ha un metodo con un nome diverso
  - ToIntFunction ha applyAsInt, Function ha apply
  - **retrocompatibilità** con le "vecchie functional interfaces"
  - **facilita il type inference** avere f.applyAsInt invece di f.apply

Lambda Expression

parameter_list -> expression | block

metodo anonimo, senza throws, tipi inferiti dal compilatore

Lambda Expression

```
button.addActionListener(event ->
       JOptionPane.showMessageDialog(frame,"you clicked!"));
```

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    JOptionPane.showMessageDialog(frame, "you clicked!");
  }
});
```

Functional Interface

Lambda Expression

parameter_list -> expression | block

metodo anonimo, senza throws, tipi inferiti dal compilatore

valuta il contesto d'uso cercando di inferire i tipi di Input/Output e vede se esiste un' interfaccia funzionale corrispondente; il sistema runtime crea l'istanza dell'interfaccia

Lambda Expression

parameter_list -> expression | block

metodo anonimo, senza throws, tipi inferiti dal compilatore

- una lambda expression puo' avere **variabili libere**, definite cioe' fuori dal suo corpo
- il ***valore*** della lambda-expression e' quindi

  il lambda-termine **+** il suo *referencing environment*

  che definisce il valore delle variabili libere

cosi' si ottiene un termine chiuso:
una **closure**

Closures in Scala

```
> (x: Int) => x + more
> error: not found: value more
               (x: Int) => x + more
```

more è variabile libera!

```
> val more = 1
more: Int = 1
> val addMore = (x: Int) => x + more
addMore: Int => Int = <function1>
> addMore(10)
res: Int = 11
```

c'è contesto che dà valore a variabile libera

Closures in Scala

```
> (x: Int) => x + more
> error: not found: value more
               (x: Int) => x + more
```

more è variabile libera!

```
> var more = 1
more: Int = 1
> val addMore = (x: Int) => x + more
addMore: Int => Int = <function1>
> addMore(10)
res: Int = 11
```

c'è contesto che dà valore a variabile libera

```
> more = 999   // modifico il valore nel contesto
more: Int = 999
> addMore(10)
res: Int = 1009
```

la closure è cambiata!

Closures in Scala

```
> (x: Int) => x + more
> error: not found: value more
              (x: Int) => x + more
```

more è variabile libera!

non è una ***funzione*** ma una ***closure***!

```
> var more = 1
more: Int = 1

> val addMore = (x: Int) => x + more
addMore: Int => Int = <function1>

> addMore(10)
res: Int = 11
```

c'è contesto che dà valore a variabile libera

la closure non contiene il ***valore*** di `more` ma un ***riferimento*** a `more`

```
> more = 999   // modifico il valore nel contesto
more: Int = 999

> addMore(10)
res: Int = 1009
```

la closure è cambiata!

Closures in Java8

```
public static void main(String[] a){
  int more = 10;
  Function<Integer,Integer> f = n -> n*more;
  System.out.println(f.apply(5)); //stampa 50


  more = 7;
  System.out.println(f.apply(5));
```

more è variabile libera!

non compila!

error: local variables referenced from a lambda expression must be **final or effectively final**

```
      Function<Integer,Integer> f =n -> n*more;
                                        ^
```


Closures in Scala

```
> val someNumbers = List(-11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

> var sum = 0
sum: Int = 0

> someNumbers.foreach(sum += _)

> sum
res: Int =-11
```

`x => { sum +=x }`

la chiamata della closure modifica il valore di `sum`!!

NON E' BUONA PROGRAMMAZIONE FUNZIONALE!!
…no side effects! e usare sempre `vals`

Closures in Java8

```
> someNumbers.foreach( x => {sum += x} )
```

```
static void doFor(Integer[] array, Consumer<Integer> c){
      for(int i : array)
          c.accept(array[i]);
   }

public static void main(String[] a){
      Integer[] arr = {1,2,3,4,5};
      int sum=0;
      Consumer<Integer> c = x -> {sum += x; };
      doFor(arr,c);
      System.out.println(sum);
   }
```

non compila!

error: local variables referenced from a lambda expression must be **final or effectively final**

Closures

Le variabili libere di una funzione/lamda expr

- devono essere definite nello scope in cui è definita la lambda expr,
  - In **Java8** devono essere "***effectively immutable***" (non necessariamente **`final`**)
  - in **Scala** possono essere **`val`** o **`var`**
- il valore runtime di una funzione/lambda expr è una **closure**, cioè il *termine + il riferimento al suo environment*
  - in **Java8** è sufficiente che la closure catturi il valore non il riferimento alla variabile libera

C++11 lambdas

[]	Capture nothing
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else
[this]	Capture the this pointer of the enclosing class

Exceptions and lambda expr in Java8

- functional interfaces (usually) define a method that does not throws Exceptions (e.g. apply, accept, test…)
- therefore lambda expressions **must**:
  - either **use try-catch block to locally handle exceptions**
  - or locally wrap exceptions as **`RuntimeExceptions`** that flows upward….horrible!!

….Scala offers the **`Try[T]`** type!

Lambda Expressions in Java8 - scope

- **regole di scope:**
  - le lambda expr **annidate in una *classe*** hanno un proprio scope (i.e. nomi li' dichiarati nascondono gli stessi nomi dichiarati nello scope esterno)
  - le lambda expr **annidate in un *metodo*** non hanno un proprio scope ma sono parte dello scope del metodo in cui stanno
  - se una lambda expr si riferisce a variabile locale definita nel metodo in cui e' annidata, allora tale variabile deve essere **`final`**

Method References

- una lambda expression codifica una "*funzionalità*"
- anche i metodi delle classi lo fanno

  ⟹ possiamo usare dei metodi dove ci si aspetta una lambda expr
- come si indicano? come ***passare un metodo come parametro***?
  - **method reference**: sono handle a metodi esistenti:
  - metodi statici **`Type::staticMethName`**
  - medodi di istanza di uno specifico oggetto **`reference::methName`**
  - metodi di istanza di un qualsiasi oggetto di un certo tipo **`Type::methName`**

esempio:

```
@FunctionalInterface
public interface IntBinaryOperator {
     int applyAsInt(int left, int right);
}
```

Java <8

```
IntBinaryOperator b1 = new IntBinaryOperator() {
       public int applyAsInt(int left, int right) {
              return left + right;
       };
System.out.println(b1.applyAsInt(10,10));
```

lambda expression

```
IntBinaryOperator b2 = (left, right) -> left + right;
System.out.println(b2.applyAsInt(10,10));
```

per applicare la lambda expr
va sempre ***invocato esplicitamente***

Java <8

```
IntBinaryOperator b1 = new IntBinaryOperator() {
       public int applyAsInt(int left, int right) {
              return left + right;
       };
System.out.println(b1.applyAsInt(10,10));
```

lambda expression

```
IntBinaryOperator b2 = (left, right) -> left + right;
System.out.println(b2.applyAsInt(10,10));
```

per applicare la lambda expr
e ***invocato esplicitamente***

sta per la lambda
`(left,right) -> Integer.sum(left,right)`

riferimento a metodo statico

```
IntBinaryOperator b3 = Integer::sum;
System.out.println(b3.applyAsInt(10,10));
```

va sempre ***invocato esplicitamente***

S.Crafa UniPd AA17/18

riferimento a metodo di istanza di un oggetto

```
class MyInteger {
  public int mySum(int left, int right){return left+right;}
}

MyInteger mi = new MyInteger();
IntBinaryOperator b4 = mi::mySum;   // (l,r)-> mi.mySum(l,r)
System.out.println(b4.applyAsInt(10,10));
```

riferimento a metodo di un qualsiasi oggetto di un tipo

```
interface MyIntBinaryOp {
  int applyAsInt(MyInteger m, int a, int b);
}
MyIntBinaryOp b5 = MyInteger::mySum;
System.out.println(b5.applyAsInt(new MyInteger(),10,10));
```

tradotto in
`(MyInteger m, int l, int r)-> m.mySum(l,r)`
quindi non ha tipo `IntBinaryOPperator` ma `MyIntBinaryOp!!`

method reference

```
Function<String,String> f = String::toLowercase;
f.apply("PIPPO");
```

: String -> String

metodo non statico di `String`
: () -> String

- inizializzato con un method reference a metodo di istanza che chiede un oggetto `String` di invocazione (che sara' fornito al momento di fare `apply`)
- `String::toLowercase()` *: String -> String = Function<S,S>*
- e la stringa su cui invocare `toLowercase` e' quella passata ad `apply`

quindi il typing di `f` e' corretto!!

…i metodi non sono funzioni!

S.Crafa UniPd AA17/18