Espressioni Lambda

Funzioni anonime

Lambda functions

A block of code that you can pass around as so it can be executed later, once or multiple times.

Lambda syntax:

(parameters) -> {lambda body}

- small one or two line functions
- A lambda expression is an anonymous function, and it is not associated with a class.
- Reduce verbosity caused by anonymous classes and need for inner classes

- Alonzo Church in his invention of the lambda calculus in 1936.
- Lisp 1958
- Supported in C#, JavaScript, Python, Ruby, C++11

Lambda Syntax

```
//argument list
() -> System.out.println("Hello Lambda") ;
() ->{throw new RuntimeException()}
x -> x+10;
(int x, int y) -> { return x + y; };
(String x, String y) -> x.length() - y.length();
(x, y) -> x.length() - y.length();

//single expressions
x -> {x>=0 ? True: False}
()-> 42

//list of statements
(String x) -> {
  listA.add(x);
  listB.remove(x);
  return listB.size();
  }
```

How are they different from Java methods?

- Argument type deduction
- Local variable capturing
- Omission of brackets and return statement
- Return type deduction

Vedi VSCODE lambda/intro

Functional Interfaces

- An interface that has only one abstract method.
  - Before Java 8 this was obvious, only one method.
  - Previously known as SAM's, Single Abstract Methods
  - Were used in java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator, java.util.concurrent.Callable
  - Java 8 introduced Default methods…
  - Now use @FunctionalInterface annotation (may be omitted)
  - Generates compiler error when there is more than one abstract method

Package java.util.function

- Well defined set of general purpose functional interfaces types.
  - All have only one abstract method.
  - Lambda expressions can be used wherever these types are referenced.

```
new Thread(()-> System.out.println("Hello World!")).start();
```

  - Inject functionality into methods, the same way we inject values into methods!
  - Used extensively in the Java class libraries. Especially with the Stream API.
  - Variations on number of arguments and specific for primitive types.

Functional Interfaces

- Attributing a lambda expression to a variable of Functional Interface Type
- The lambda expression provides the implementation of the abstract method.
- Returning a lambda expression is also possible

Example

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

Runnable r = ()->
    System.out.println("Hello World!");
new Thread(r).start();
```

```
//functional interface implementation
List<String> strs = ...;
Collections.sort(strs, (s1, s2) ->
    Integer.compare(s1.length(), s2.length()));

            new Thread(() -> {
                        connectToService();
                        sendNotification();
                    }).start();
```

Vedi vscode lambda/functionalInterfaces

Package java.util.function

- Predicate<T> - a boolean-valued property of an object. [boolean test()]
- Consumer<T> - an action to be performed on an object. [void accept(T)]
- Function<T,R> - a function transforming a T to a R. [R apply(T)]
- Supplier<T> - provide an instance of a T (such as a factory) [T get()]
- UnaryOperator<T> - a function from T to T. [T apply(T)]
- BinaryOperator<T> - a function from (T,T) to T- More meaningful type names

Vedi esempio su VSCode: lambda/syntax

Method & Constructors References

Treating an existing method as an instance of a Functional Interface

- Object oriented way of attributing a method to a variable
- :: operator

```
class Person {
    private String name;
    private int age;

    public int getAge() {return this.age;}
    public String getName() {return this.name;}
}

Person[] people = ...;
Comparator<Person> byName = Comparator.comparing(Person::getName);
Arrays.sort(people, byName);
```

External Variables

Lambda expressions can refer to variables from the surrounding scope.

```
class DataProcessor {
   private int currentValue;
     public void process(int par) {
      int n = 1000; // eff. Final
      DataSet myData = myFactory.getDataSet(n++);
      myData.forEach(d -> d.use(par, n, this.currentValue++));
     }
}
```

- Static and instance variable are ok. Only local variables must be Effectively Final

  Effectively final: assigned only once, even if not explicitly declared final.

- 'this' refers to the enclosing object, not the lambda itself.
- Remember the Lambda is not associated with a class, therefore there can be no 'this' for a lambda.

Why Default Methods?

Add default behaviours to interfaces

Java 8 has lambda expressions.

List<?> list = ...

list.forEach(/* lambda code goes here */);

```
@FunctionalInterface
public interface Iterable {
   default void forEach(Consumer<? super T>
action) {
         for (T t : this) {
             action.accept(t);
         }
     }
}
```

There was a problem: can't use them…

Changing existing interfaces (java.util.List, java.util.Collection) would break their current library implementations

Solution: Java 8 introduce default methods! Existing libraries inherit the default implementation…

forEach(consumer<T>) is one default method in new Iterable<T> functional interface

New default interface methods in Java 8

Interface	Description
`Iterable.forEach(Consumer c)`	`myList.forEach(System.out::println);`
`Collection.removeIf(Predicate p)`	`myList.removeIf(s -> s.length() == 0);`
`List.replaceAll(UnaryOperator o)`	`myList.replaceAll(String::toUpperCase);`
`List.sort(Comparator c)`	`myList.sort((x, y) -> x.length() – y.length());`

Replaces Collections.sort(List l, Comparator c)

Default methods can be seen as a bridge between lambdas and existing JDK libraries.

Multiple inheritance in Java 8 ! Defaults can cause conflict of behaviour…

Conflict resolution

```
// Java program to demonstrate the case
// when two interfaces are overridden

// Creating Interface One
interface GfG{
    public default void display()
    {
        System.out.println("GEEKSFORGEEKS");
    }
}

// Creating Interface Two
interface gfg{

    public default void display()
    {
        System.out.println("geeksforgeeks");
    }
}
```

```
public class InterfaceExample implements GfG,gfg {

    // Interfaces are Overrided
    public void display() {

        GfG.super.display();

        gfg.super.display();
    }

    public static void main(String args[]) {
        InterfaceExample obj = new InterfaceExample();
        obj.display();
    }
}
```

Lambdas summary

- CPU's are getting much faster, but we are getting more cores to use.
- Inject functionality into methods, the same way we inject values into methods.
- Functional interfaces are interfaces that have only one abstract method.
- Lambdas can be used where the type is a functional interface.
- Use lambdas as method parameters or assign them to variables.
- The java.util.function package already gives us a few functional interfaces out of the box.
- Method and constructor references as short hands notation, **::** (double colon)
- There are a few new methods that can use lambdas in Java 8, e.g. `forEach()`

15 min. break