Scala vs Java8

- Tipi

```
Int => Double
```
sta per
```
trait Function1[Int,Double]{
 def apply(x:Int):Double
}
```
implementazone uniforme ma sintassi di tipi agevole e chiara

```
interface Function<Integer,Double>{
    Double apply(x:Integer);
}
```
sintassi verbosa…
`IntBynaryOperator` con `applyAsInt`
`Predicate<T>` con `test`
…bisogna conoscere i nomi dei tipi e dei metodi!!

- Funzioni

lambda expr, ma anche **altro per integrarle con stile funzionale** (e.g. pattern matching)

N.B. I paramteri sono `vals`

lambda expr con sintassi agevole, **anche method references** (con sottigliezze)

N.B. I parametri sono mutable

Scala vs Java8

- Tipi

```
Int => Double
```
sta per

incentiva a **PASSARE** funzioni e non a *INVOCARE* funzioni…

```
interface Function<Integer,Double>{
    Double apply(x:Integer);
}
```
sintassi verbosa…
`IntBynaryOperator` con `applyAsInt`
`Predicate<T>` con `test`
…bisogna conoscere i nomi dei tipi e dei metodi!!

lambda expr, ma anche **altro per integrarle con stile funzionale** (e.g. pattern matching)

N.B. I paramteri sono `vals`

lambda expr con sintassi agevole, **anche method references** (con sottigliezze)

N.B. I parametri sono mutable

Esempio di codice funzionale in Java8

- lavorare con una risorsa facendo rispettare una policy

```
T t = setUp();
…operations on t…
cleanUp();
log();
```

The operations to perform are mixed with the policy-enforcing code

What about exception?

```
doOnT(T t -> {operations on t});
```

```
void doOnT(Consumer<T> f){
  f.accept(setUp());
  cleanUp();
  log();
}
```

The policy-enforcing code is separated from the operations to perform (**separation of concerns**)

Easier to understand/debug/maintain

Working with resources: lock

- Explicit locks must be used according to a precise policy:

```
public class Locking {
  Lock lock = new ReentrantLock();

  public void doOp1(){
    lock.lock();
    try {
      // ... critical code ...
    } finally {
      lock.unlock();
    }
  }
```

Working with resources: lock

```
public class Locker {
  public static void runLocked(Lock lock, Runnable block){
    lock.lock();
    try {
      block.run();
    } finally {
      lock.unlock();
    }
  }
}

public void doOp2(){
  runLocked(lock, () -> { /* ... critical code … */ });
}
public void doOp3(){
  runLocked(lock, () -> { /* …other critical code… */ });
}
```

it becomes a functional interface representing a block of code to be run (not necessarily concurrently!)

Java 8 e Stream

- lo **scopo** di introdurre le **lambda expr** in Java8 e' quello di lavorare in modo funzionale/efficiente con le **collezioni**!
  - In order to operate on collection we have to deal with (multiple) **iterators** in (multiple) **for-loops** and (multiple) **mutable counters**. All these are **difficult to compose**
  - **Higher-order combiners** operating over collections **easily combine** to provide correct and **clear business logic**

Usare le collezioni

- **Iterating** through a List

```
final List<String> friends = Arrays.asList("Ada","Bob",…);
```

Java5

```
for(String name: friends){
  System.out.println(name);
}
```

iterazione esterna

che metodo e'??

Java8

```
friends.forEach(name -> System.out.println(name));
opp.
friends.forEach(System.out::println);
```

iterazione interna

Usare le collezioni

```
final List<String> friends = Arrays.asList("Ada","Bob",…);
```

```
List<T> extends Iterable<T>
```

che offre metodo

```
default void forEach(Consumer<T> action);
```

```
@FunctionalInterface
interface Consumer<T>{
 void accept(T t);
}
```

solo da Java 8
ma `Iterable<T>` c'e' da sempre:
modificate le interfacce precedenti
compila ancora il vecchio codice?

default methods

Default methods

```
interface I {
 default void foo() { …implementazione di default… }
 void boo();
}
```

- da Java 8 le interfacce possono contenere metodi concreti, definiti con il modificatore `default`
- la classe che implementa l'interfaccia ***non deve necessariamente (ri)definire questi metodi***, che vengono direttamente ereditati
- **cosi' il codice legacy che implementava le vecchie interfacce continua a funzionare!!**

ma se si implementano piu' interfacce
si possono creare **problemi di ereditarieta' multipla!**


```
interface Fly {
 default void takeOff(){…}
 default void land(){…}
 default turn(){…}
 default cruise(){…}
}
```

```
interface Sail{
 default void turn(){…}
 default void cruise(){…}
}
```

```
interface FastFly extends Fly {
 default void takeOff(){…} //override
 // eredita gli altri
}
```

```
class Vehicle{
 public void turn(){…}
}
```

```
class SeaPlane extends Vehicle implements FastFly, Sail {
 public void cruise(){
   if(...) FastFly.super.cruise();
       else Sail.super.cruise();
 }
 // turn() non e' ambiguo: x def e' quello della CLASSE Vehicle
}
```

obbligatorio ridefinire `cruise()` *per disambiguare quale metodo*

metodi statici in interfacce

```
interface I {
 default void foo() { …implementazione di default… }
 void boo();
 static void bar() { …implementaione… }
}
```

- da Java 8 le interfacce possono contenere anche metodi statici concreti
- **comodi per incapsulare metodi di utilita'**
- possono accedere solo ad altri membri statici dell'interfaccia: non metodi astratti ne' di default
- *non vengono ereditati*, ne' da sottointerfcce ne' da classi che implementano (diverso da metodi statici delle classi)

```
interface X { static void foo(){...} }
interface Y extends X {}
class A { static void foo(){...} }
class B extends A {}
```

```
X.foo(); // Ok
Y.foo(); NON COMPILA
A.foo(); // Ok
B.foo(); // Ok
```


Usare le collezioni

```
final List<String> friends = Arrays.asList("Ada","Bob",…);

friends.forEach(System.out::println);
```

n -> System.out.println(n)

: String -> void
= Consumer<String>

controlliamo che sia well typed:

```
: List<String> extends Iterable<String>

      default void forEach(Consumer<String> action);
```

Usare le collezioni

```
final List<String> friends = Arrays.asList("Ada","Bob",…);

friends.forEach(System.out::println);
```

n -> System.out.println(n)

: String -> void
= Consumer<String>

controlliamo che sia well typed:

```
List<T> extends Iterable<T>

    default void forEach(Consumer<? super T> action);

    default void forEach(Consumer<T> action);
```


Generics e Subtyping

```
…in qualche classe C…
static void log(Object obj){ …stampa su file obj.toString()… }
```

```
final List<String> friends = Arrays.asList("Ada","Bob",…);
friends.forEach(System.out::println);
friends.forEach(C::log);  // opp n -> C.log(n)
```

: Consumer<Object> $\not<:$ *Consumer<String>*

*in **Java** i generics sono **invarianti***

se `Iterable<T>` avesse un metodo con segnatura
`void forEach(Consumer<T> c);`
`friends.forEach(C::log)` **non compilerebbe**!

```
default void forEach(Consumer<? super T> action);
```


Generics e Subtyping

```
default void forEach(Consumer<? super T> action);
```

accetta parametri
di tipo `Consumer<S>` con `T <: S`

$$\frac{\texttt{T <: S}}{\texttt{S -> void <: T -> void}}$$

`Consumer<S> <: Consumer<T>`

`friends.forEach(C::log)` **compila**
perchè `C::log` di tipo `Consumer<Object>` è parametro corretto per
`void forEach(Consumer<? super String> a);`
poichè `String <: Object`

Subtyping e Generics

Java

```
class C<T> {
  public void foo(T arg){ … }
  public void boo(C<T> arg){ … }
  public void moo(C<? extends T> arg){ … }
}
```

```
C<Object> a=new C<Object>();
C<String> b= new C<String>();
a.foo("pippo"); //OK  String <: Object = T
a.boo(b); NO C<String> cannot be converted to C<Object>
a.moo(b); // OK
```

*in Java i generics sono **invarianti***

Scala

```
class C[+T] {
  public void foo(T arg){ … }
  public void boo(C[T] arg){ … }
}
```

*definito subtyping **covariante***

```
a.foo("pippo"); // OK
a.boo(b);  // OK C[String] <: C[Object]
```

Java

```
interface Consumer<T> { … }
interface Iterable<T> {
  default void forEach(Consumer<? super T> arg);
  ...
}
```

*in Java i generics sono **invarianti***

```
Iterable<String> friends = …
Consumer<String> c1 = (String n) -> System.out.print(n);
Consumer<Object> c2 = (Object o) -> C.log(o);
friends.forEach(c1);
friends.forEach(c2);
```

ok anche se `Consumer<Object>` <: `Consumer<String>`

Java

```
interface Consumer<T> { … }
interface Iterable<T> {
  default void forEach(Consumer<? super T> arg);
  ...
}
```

*in Java i generics sono **invarianti***

```
Iterable<String> friends = …
Consumer<String> c1 = (String n) -> System.out.print(n);
Consumer<Object> c2 = (Object o) -> C.log(o);
friends.forEach(c1);
friends.forEach(c2);
```

*oppure, se ci fossero notazioni di varianza, basterebbe definire `Consumer` **controvariante***

Java "polare"

```
interface Consumer<-T> { … }
interface Iterable<T> {
  default void forEach(Consumer<T> arg);
  …
}
```

così al meotdo `void forEach(Consumer<String> arg)` posso passare un `Consumer<Object>` poiche' `String <: Object` dunque `Consumer<Object> <: Consumer<String>`

Trasformare una lista

```
final List<String> friends = Arrays.asList("Ada","Bob",…);
```

```
final List<String> upperNames = new ArrayList<String>();
friends.forEach(name -> upperNames.add(name.toUpperCase()));
System.out.println(upperNames);
```

MA richiede una lista in piu' da modificare

```
friends.stream()
       .map(name -> name.toUpperCase())
       .forEach(name -> System.out.print(name + " "));
```

`.map(String::toUpperCase())`

outputs another stream (not a collection!)

Stream

- uno **stream** e' un'astrazione che rappresenta una **sequenza di elementi su cui e' possibile compiere operazioni aggregate**, sia sequenziali che parallele
- **collezione ≠ stream**

Pipeline di Stream

- una computazione su un insieme di elementi di una collezione e' eseguita mediante la costruzione di una **pipeline di stream**, fatta da:
  - **una sorgente**: una collezione, un array, un canale di I/O, una funzione generatrice
  - zero o piu' **operazioni intermedie**
  - **un'operazione terminale**

definite da lambda expr, o method ref.

Stream

- uno **stream** e' un'astrazione che rappresenta una **sequenza di elementi su cui e' possibile compiere operazioni aggregate**, sia sequenziali che parallele

- **collezione ≠ stream**
  - da collezione a stream: inserito in `interface Collection<E>`

    `default Stream<E> stream()`

  - da stream a collezione: ***(mutable) reduction operations***, che accumulano degli elementi di input in un valore finale o in un container finale, eventualmente con una trasformazione finale.

    e.g. `max, count, sum, average` …

    Queste operazioni possono essere *sequenziali* o *parallele*

Trasformare una lista

```
final List<String> friends = Arrays.asList("Ada","Bob",…);
```

```
friends.stream()
   .map(String::toUpperCase())                  // op intermedia
   .forEach(name->System.out.print(name+" "));//op terminale
```

Finding an element in a list

```
final List<String> friendsWithN =
  friends.stream()
         .filter(name -> name.startsWith("N")) // intermedia
         .collect(Collectors.toList());        // termiale
```

ritorna uno stream

ritorna una lista: la lista dei nomi che iniziano con N

Typing - map

dentro `interface Stream<T>`

```
Stream<R> map(Function<? super T,? extends R> mapper)
```

su uno stream di valori `T` applica ad ogni elemento una funzione `T -> R` e ritorna uno stream di valori di tipo `R`

o un sottotipo, cioè `T1 -> R1` con `T <: T1` e `R1 <: R`

```
friends.stream()          : Stream<String>
       .map(String::toUpperCase())   : Stream<String>
```

: Function<String,String>

Typing - filter

dentro `interface Stream<T>`

`Stream<T> filter(Predicate<? super T> predicate)`

dato uno stream di valori `T` ritorna uno stream con quei valori di tipo `T` che soddisfano il predicato: cioe' una funzione `T -> bool`

o **un sottotipo,** cioè `T1 -> bool`, con `T <: T1`

```
friends.stream()                                  : Stream<String>
      .filter(name -> name.startsWith("N"))       : Stream<String>
                                                  : String -> bool = Predicate<String>
```


Trovare un elemento

- torvare nella lista il primo nome che inizia con una data lettera

```
public static void pickname(
         final List<String> names, final String starting) {
   String foundName = null;
   for(String name : names){
     if(name.startsWith(starting)){
       foundName = name;
       break;
     }
   }
   if(foundName != null)
     System.out.println(foundName + "starts with"+ starting);
   else System.out.println("not found");
}
```

using null we must then check the reference to avoid NullPointerExc.

Trovare un elemento

- torvare nella lista il primo nome che inizia con una data lettera

```
public static void pickname(
         final List<String> names, final String starting) {

final Optional<String> foundName =
    names.stream()
         .filter(name -> name.startsWith(starting))
         .findFirst();

  System.out.println(
     String.format("name starting with %s: %s",
                starting, foundName.orElse("not found")));
}
```

lazy implementation
si ferma al primo che incontra

metodi di `Optional<T>` :
`get(), isPresent(), orElse(T t), ifPresent(Consumer<T> c), …`

Operazioni di riduzione

- torvare la somma delle lunghezze dei nomi nella lista

```
final List<String> friends = Arrays.asList("Ada","Bob",…);

final int sumLen = friends.stream()
                      .mapToInt(name -> name.length())
                      .sum();
           oppure     .reduce(0, (a,b)->a+b );
```

nella classe `Stream<T>` c'e' il metodo

`T reduce (T identity, BinaryOperator<T> accumulator)`

`reduce(identiy, accumulator)`

```
T result = identity;
for(T elem : stream)
  result = accumulator.apply(result, elem);
```

Operazioni di riduzione

- stampa il nome piu' lungo, scorrendo elenco una sola volta

```
final Optional<String> longName =
  friends.stream()
         .reduce((name1,name2) ->
              name1.length() >= name2.length() ? name1 : name2);

longName.ifPresent(name -> System.out.println(name));
```

nella classe **`Stream<T>`** c'e' anche il metodo

```
Optional<T> reduce (BinaryOperator<T> accumulator)
```


Pipeline di Stream

- **una sorgente**: una collezione, un array, un canale di I/O, una funzione generatrice
- zero o piu' **operazioni intermedie**
- **un'operazione terminale**

Map-Reduce Pattern

- producono un ***nuovo stream***
- sono ***lazy***, cioe' non sono immediatamente eseguite
- ogni nuovo elemento puo' essere processato indipendentemente dagli altri (*stateless*) oppure no (*stateful*)

- producono un ***risultato***, una collezione o niente, ma non uno stream
- sono ***eager***, cioe' attraversano subito la sorgente per effettuare le operazioni della pipeline
- al termine la pipeline non si puo' riutilizzare

eager vs lazy evaluation

- eager code is easier to write and to reason about, but lazy code sometimes can be more efficient:
  - e.g. postponing the creation of heavyweight objects or executing expensive computations until we really need it
  - **`fn1() || fn2()`** **`fn1() && fn2()`**

```
boolean eagerAnd(boolean x, boolean y) {
     return x && y;
}
eagerAnd(heavyMeth(1),heavyMeth(2));          : void -> Boolean

boolean lazyAnd(Supplier<Boolean> x, Supplier<Boolean> y) {
    return x.get() && y.get();
}
lazyAnd(()->heavyMeth(1), ()->heavyMeth(2));
```


Lazy evaluation of Streams

- Streams have two types of methods:
  - ***intermediate***, like **`map()`** or **`filter()`**: calls to them return immediately and their core behavior (encoded by the lambda-expr passed as parameters) is cached for later execution
  - ***terminal*** method, like **`findFirst()`** or **`reduce()`**: their call cause (some) cached behavior to be run. The computation will complete as soon as the desired result is found, therefore *not all cached code is executed for sure*.

```
List<String> friends = Arrays.asList("Brad","Kate","Kim"...};

final Optional<String> threeLetters =
     friends.stream()
          .filter(name -> name.length()==3)
          .map(name -> name.toUpperCase())
          .findFirst();
```

change this lambda so to trace their execution and have a look!

it does not apply filter and map to all names; it just processes an element at the time throughout the whole chain

parallelStream()

```
final List<String> friends = Arrays.asList("Ada","Bob",…);

friends.stream()
       .map(name -> name.toUpperCase())
       .forEach(name -> System.out.print(name + " "));
// prints ADA BOB ...
```

```
friends.parallelStream()
       .map(name -> name.toUpperCase())
       .forEach(name -> System.out.print(name + " "));
// names may be printed in any order
// different executions, possibly different orders
```

```
friends.parallelStream()
       .map(name -> name.toUpperCase())
       .forEachOrdered(name -> System.out.print(name + " "));
// prints ADA BOB...  still concurrent but less efficient
```


parallelStream()

- it manages methods like **`map()`** and **`filter()`** in parallel across multiple threads, ***managed by a pool of threads under the hood***.
- But consider:
  - do we really want to run the lambda expr concurrently?
  - **the code should be able to run independently without causing any side effects or race conditions *(avoid mutability and use pure functions)***
  - the **correctness of the solution should not depend on the order of execution of the lambda exprs** that are scheduled to run concurrently.
- **make sure that the time savings far outweighs the cost of using concurrency (sufficiently large collections)**

Infinite, lazy, collections

- the infinite series of prime numbers

```
public class Primes{

  public static boolean isPrime(final int number) {
    return number >1 &&
      IntStream.rangeClosed(2, (int)Math.sqrt(number))
               .noneMatch(divisor -> number%divisor ==0);
  }


  // returns the first prime >number
  private static int primeAfter(final int number) {
    if(isPrime(number+1))
      return number+1;
    else
      return primeAfter(number+1);
  }
```

returns a closed interval of `int` as a `Stream`

it is a **`UnaryOperator<T>`** i.e. **`T->T`**, that we can use to produce the next prime

Infinite, lazy, collections

```
static Stream<T> iterate(T seed, UnaryOperator<T> f)
```

Returns an ***infinite*** sequential ordered `Stream` produced by iterative application of a function **`f`** to an initial element **`seed`**, i.e. a **`Stream`** consisting of `seed, f(seed), f(f(seed)), ...`

```
static List<Integer> primes(final int from, final int count) {
  return Stream.iterate(primeAfter(from-1), Primes::primeAfter)
               .limit(count)
               .collect(Collectors.<Integer>toList()));
}
```

they are **both intermediate operations**, that **lazily notes the number of elements** needed for later evaluation!

Returns a stream consisting of the elements of this stream, truncated to be no longer than **`count`** in length.