17
Lambda expressions

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

Goal della lezione

- Illustrare il concetto di lambda
- Dettagliare il supporto alle lambda (da Java 8+)

Argomenti

- Lambda expressions
- Functional interfaces
- Altri usi nell'API

Outline

Programmazione funzionale in Java

Introdotta da Java 8 (2014)

- Un significativo cambio di direzione nella OOP "mainstream"
- Principale novità: lambda (ossia uno degli elementi fondamentali dello stile di programmazione funzionale)
- Le lambda portano ad uno stile più elegante e astratto di programmazione
- In Java, portano a codice più compatto e chiaro in certe situazioni
- Impatta alcuni aspetti di linguaggio, varie librerie
- Può impattare significativamente il nostro "stile" di programmazione

Risorse

- Libri: R.Warburton, Java 8 Lambdas
- Tutorial in rete:

  `http://www.techempower.com/blog/2013/03/26/everything-about-java-8/`
- Specification: `http://cr.openjdk.java.net/~dlsmith/jsr335-0.6.1/`

Preview 1: strategie funzionali

```java
public class FirstComparable {

  public static void main(String[] args) {

    final List<Person> list = new ArrayList<Person>();
    list.add(new Person("Mario",1960,true));
    list.add(new Person("Gino",1970,false));
    list.add(new Person("Rino",1951,true));
    System.out.println(list);

    // Without lambdas
    Collections.sort(list, new Comparator<Person>(){

      public int compare(Person o1, Person o2) {
        return o1.getBirthYear() - o2.getBirthYear();
      }

    });
    System.out.println(list);

    // With lambdas
    Collections.sort(list, (o1,o2) -> o2.getBirthYear() - o1.getBirthYear())
     ;
    System.out.println(list);
  }

}
```

Handler eventi senza lambda

```
1  public class UseButtonEvents{
2
3    public static void main(String[] args){
4      final JButton b1 = new JButton("Say Hello");
5      b1.addActionListener(new ActionListener(){
6        public void actionPerformed(ActionEvent e) {
7          System.out.println("Hello!! "+e);
8        }
9      }); // Uso una inner class anonima..
10
11     final JButton b2 = new JButton("Quit");
12     b2.addActionListener(new ActionListener(){
13       public void actionPerformed(ActionEvent e) {
14         System.out.println("Quitting.."+e);
15         System.exit(0);
16       }
17     }); // Uso una inner class anonima..
18
19     final JFrame frame = new JFrame("Events Example");
20     frame.setSize(320, 200);
21     frame.getContentPane().setLayout(new FlowLayout());
22     frame.getContentPane().add(b1);
23     frame.getContentPane().add(b2);
24     frame.setVisible(true);
25   }
26 }
```

Preview 2: handler eventi con le lambda

```
1  public class UseButtonEventsWithLambda{
2
3    public static void main(String[] args){
4      final JButton b1 = new JButton("Say Hello");
5      // lambda con single-expression body
6      b1.addActionListener( e -> System.out.println("Hello!!"+e));
7
8      final JButton b2 = new JButton("Quit");
9      // lambda con body
10     b2.addActionListener( e -> {
11       System.out.println("Quitting.."+e);
12       System.exit(0);
13     });
14
15     final JFrame frame = new JFrame("Events Example");
16     frame.setSize(320, 200);
17     frame.getContentPane().setLayout(new FlowLayout());
18     frame.getContentPane().add(b1);
19     frame.getContentPane().add(b2);
20     frame.setVisible(true);
21   }
22 }
```

Preview 3: iterazioni "dichiarative" con gli stream

```java
public class FirstStream {

  public static void main(String[] args) {
    final List<Integer> list = List.of(10,20,30,40,50,60,70,80,90);

    // for each element, pass it to method println of System.out
    list.stream().forEach(System.out::println); //"10" \n "20" \n "30"

    // Filter and print
    list.stream().filter(i->i>60).forEach(i -> System.out.print(i+" "));
    System.out.println(""); //"70 80 90 "

    // Map and print
    list.stream().map(i->i+" ").forEach(System.out::print);
    System.out.println(""); //"10 20 .. 70 80 90 "

    // Map-reduce and print the resulting string
    final String s = list.stream().map(i->i+"|").reduce((x,y) -> x+y).get();
    System.out.println(s); //"10|20|..|70|80|90|"

  }

}
```

Outline

Elementi delle lambda expression

Che cos'è una lambda

- è una funzione (anonima) con accesso ad uno scope locale
- è applicabile a certi input, e dà un risultato (oppure `void`)
- per calcolare il risultato potrebbe usare qualche variabile nello scope in cui è definita – stesso scope delle classi anonime
- la lambda è usabile come "valore" (quindi, come dato), ossia è passabile a metodi, altre funzioni, o memorizzata in variabili/campi
- ossia si può "passare" del "codice"

Caratteristica specifica di Java

- come vedremo, una lambda è un oggetto, ed è passabile là dove ci si aspetta una `interface` detta "funzionale"
- metodi statici o istanza possono essere usati a mo' di lambda (chiamati "method reference"), perché sono interpretabili come funzioni (come i `delegate` di `C#`)

Come si esprime una lambda

Sintassi possibili

- `(T1 x1,..,Tn xn) -> {<body>} // sintassi completa`
- `(x1,..,xn) -> {<body>} // completa non tipata`
- `x -> {<body>} // argomento singolo`
- `(T1 x1,..,Tn xn) -> <exp> // expression body`
- `(x1,..,xn) -> <exp> // non tipato con expr.  body`
- `x -> <exp> // expression body e singolo argomento`
- .. oppure un "method reference"

Ossia, anche in combinazione:

- Per gli argomenti si può esprimere un tipo o può essere inferito
- Con un argomento, le parentesi tonde sono omettibili
- Il body può essere direttamente una singola espressione/istruzione

Esempi di Lambda

```java
public class AllLambdas {

  private static int mycompare(final String a, final String b) {
    return a.compareTo(b);
  }

  public static void main(String[] args) {
    final List<String> list = List.of("a", "bb", "c", "ddd");

    // sintassi completa
    Collections.sort(list, (String a, String b) -> {
      return a.length() - b.length();
    });
    System.out.println(list); // [a, c, bb, ddd]

    // sintassi con type inference
    Collections.sort(list, (a, b) -> {
      return a.length() - b.length();
    });
    System.out.println(list); // [a, c, bb, ddd]

    // sintassi con single-expression body
    Collections.sort(list, (a, b) -> a.length() - b.length());
    System.out.println(list); // [a, c, bb, ddd]

    // il body è una chiamata di metodo diretta: userò un method reference
    Collections.sort(list, (a, b) -> mycompare(a, b));
    System.out.println(list); // [a, c, bb, ddd]
```

I method reference

Sintassi possibili

1. `<class>::<static-method>`
   - sta per `(x1,..,xn)-> <class>.<static-method>(x1,..,xn)`
2. `<class>::<instance-method>`
   - sta per `(x1,x2,..,xn)-> x1.<instance-method>(x2,..,xn)`
3. `<obj>::<method>`
   - sta per `(x1,..,xn)-> <obj>.<method>(x1,..,xn)`
4. `<class>::new`
   - sta per `(x1,..,xn)-> new <class>(x1,..,xn)`

Method reference:

- Corrisponde ad una lambda che di fatto è realizzata da un metodo
- Si può riferire un metodo statico o non, o un costruttore
- Usabile "naturalmente" quando la lambda non fa altro che chiamare un metodo usando "banalmente" i suoi input e restituendo il suo "output"
- Usarli dove possibile!

Esempi di Method Reference (casi 1,2,3)

```java
public class MethodReferences {

  private static int staticMyCompare(final String a, final String b) {
    return a.compareTo(b);
  }
  private int instanceMyCompare(final String a, final String b) {
    return b.compareTo(a);
  }
  public static void main(String[] args) {
    final List<String> list = List.of("a", "bb", "c", "ddd");
    final MethodReferences objAL = new MethodReferences();

    Collections.sort(list, (x,y) -> staticMyCompare(x,y));
    Collections.sort(list, MethodReferences::staticMyCompare);  // same as
     above
    System.out.println(list); // [a, bb, c, ddd]

    Collections.sort(list, (x,y) -> objAL.instanceMyCompare(x, y));
    Collections.sort(list, objAL::instanceMyCompare);  // same as above
    System.out.println(list); // [ddd, c, bb, a]

    Collections.sort(list, (x,y) -> x.compareTo(y));
    Collections.sort(list, String::compareTo);  // same as above
    System.out.println(list); // [ddd, c, bb, a]
  }
}
```

Dove si può usare una lambda?

Definizione di interfaccia "funzionale" (def. non definitiva)

- È una `interface` con un singolo metodo

Quale tipo è compatibile con una lambda?

- Una lambda può essere passata dove ci si attende un oggetto che implementi una interfaccia funzionale
- C'è compatibilità se i tipi in input/output della lambda (inferiti o non) sono compatibili con quelli dell'unico metodo dell'interfaccia

Motivazione:

- Di fatto, il compilatore traduce la lambda nella creazione di un oggetto di una classe anonima che implementa l'interfaccia funzionale
- Uno specifico opcode a livello di bytecode evita di costruirsi effettivamente un `.class` per ogni lambda, ma è solo una ottimizzazione interna

Generazione automatica della classe anonima

```java
public class FirstComparable2 {

  public static void main(String[] args) {

    final List<Person> list = new ArrayList<Person>();
    list.add(new Person("Mario",1960,true));
    list.add(new Person("Gino",1970,false));
    list.add(new Person("Rino",1951,true));
    System.out.println(list);

    // Sorting with a lambda
    Collections.sort(list, (p1,p2) -> p2.getBirthYear()-p1.getBirthYear());
    System.out.println(list);

    // Nota che sort richiede un Comparator<Persona>, che ha il solo metodo:
    //    int compare(Persona p1, Persona p2)
    // Quindi il codice equivalente generato da javac è:
    Collections.sort(list, new Comparator<Person>(){
      public int compare(Person p1, Person p2) {
        return p2.getBirthYear()-p1.getBirthYear();
      }
    });

    System.out.println(list);
  }

}
```

Esempio: funzione riusabile di filtraggio

```java
public interface Filter<X> {

  // Does element x pass the filter?
  boolean applyFilter(X x);

}
```

```java
public class FilterUtility {

  public static <X> Set<X> filterAll(Collection<X> set, Filter<X> filter){
    final Set<X> newSet = new HashSet<>();
    for (final X x: set){
      if (filter.applyFilter(x)){
        newSet.add(x);
      }
    }
    return newSet;
  }

  public static void main(String[] args) {
    final List<Integer> ls = List.of(10,20,30,40,50,60);

    // Nota che il nome del metodo in Filter non è mai menzionato qui
    System.out.println(filterAll(ls,x -> x>20)); // [30,40,50,60]
    System.out.println(filterAll(ls,x -> x>20 && x<60)); // [30,40,50]
    System.out.println(filterAll(ls,x -> x%20==0)); // [20,40,60]
  }
}
```

Lambda che accedono al loro scope

```
1  public class ChangeButton extends JFrame{
2
3    private int counter = 0;
4
5    public ChangeButton(){
6      super("Changing button example");
7      final JButton button = new JButton("Val: ");
8      button.addActionListener(
9        e -> button.setText("Val: " + ChangeButton.this.counter++));
10     // button è una variabile finale nel metodo
11     // ChangeButton.this.counter è un campo nella enclosing instance
12
13     this.setSize(320, 200);
14     this.getContentPane().setLayout(new FlowLayout());
15     this.getContentPane().add(button);
16     this.setVisible(true);
17   }
18
19   public static void main(String[] args){
20     new ChangeButton();
21   }
22 }
```

Metodi `default` nelle interfacce

Da Java 8 è possibile fornire implementazioni ai metodi delle `interface`

- sintassi: `interface I { ... default int m(){...}}`
- significato: non è necessario implementarli nelle sottoclassi
- .. è possibile avere anche metodi statici

Utilità

- consente di aggiungere metodi a interfacce senza rompere la compatibilità con classi esistenti che le implementano
- fornire "comportamento" ereditabile in modalità multipla
- costruire più facilmente interfacce funzionali: queste devono in effetti avere un unico metodo senza default
- consente di realizzare il patter template method solo con interfacce

Esempi di interfacce con metodi di default

- `Iterable`, `Iterator`, `Collection`, `Comparator`

Esempio SimpleIterator

```java
public interface SimpleIterator<X> {

  X getNext();

  // a template method: i.e., it calls abstract getNext()
  default List<X> getListOfNext(int n){
    final List<X> l = new LinkedList<>();
    for (int i = 0; i < n; i++){
      l.add(getNext());
    }
    return l;
  }

}
```

```java
public class UseSimpleIterator {

  public static void main(String[] args){
    final Counter c = new Counter();
    final SimpleIterator<Integer> si =
        () -> {c.inc(); return c.getValue();};

    System.out.println(si.getNext());
    System.out.println(si.getListOfNext(20));
  }

}
```

Annotazione `@FunctionalInterface`

Uso

- da usare (opzionalmente, ma consigliata!) per interfacce funzionali
- il compilatore la usa per assicurarsi che l'interfaccia sia funzionale, ossia che vi sia un solo metodo "astratto"
- per l'utilizzatore, lo si avverte che si possono usare delle lambda come "implementazione"
- nella Java API viene usata spesso

Esempio SimpleIterator

```
1  @FunctionalInterface
2  public interface SimpleIterator2<X> {
3
4    X getNext();
5
6    default List<X> getListOfNext(int n){ // template method
7      final List<X> l = new LinkedList<>();
8      for (int i = 0; i < n; i++){
9        l.add(getNext());
10     }
11     return l;
12   }
13
14 }
```

Outline

Interfacce funzionali di libreria – package `java.util.function`

Perché scriversi una nuova interfaccia funzionale all'occorrenza?

- Lo si fa solo per rappresentare concetti specifici del dominio
- Lo si fa se ha metodi default aggiuntivi
- Altrimenti, è bene usare interfacce di libreria "note"

In `java.util.function` vengono fornite varie interfacce "general purpose"

- Sono tutte funzionali
- Hanno metodi aggiuntivi default di cui non ci occupiamo
- Hanno un metodo "astratto" chiamato, a seconda: `apply`, `accept`, `test` o `get`; è quello da chiamare per applicare la funzione

Package java.util.function

Interfacce base

- `Consumer<T>: accept:(T)->void`
- `Function<T,R>: apply:(T)->R`
- `Predicate<T>: test:(T)->boolean`
- `Supplier<T>: get:()->T`
- `UnaryOperator<T>: apply:(T)->T`
- `BiConsumer<T,U>: accept:(T,U)->void`
- `BiFunction<T,U,R>: apply:(T,U)->R`
- `BinaryOperator<T>: apply:(T,T)->T`
- `BiPredicate<T,U>: test:(T,U)->boolean`
- `java.lang.Runnable: run:()->void`

Altre interfacce (usano i tipi primitivi senza boxing)

- `BooleanSupplier: get:()->boolean`
- `IntConsumer: accept:(int)->void`
- ...

Esempio: funzione riusabile di filtraggio via Predicate

```
1  import java.util.*;
2  import java.util.function.Predicate;
3
4  public class FilterUtility2 {
5
6    public static <X> Set<X> filterAll(Collection<X> set,Predicate<X> filter){
7      final Set<X> newSet = new HashSet<>();
8      for (final X x: set){
9        if (filter.test(x)){
10         newSet.add(x);
11       }
12     }
13     return newSet;
14   }
15
16   public static void main(String[] args) {
17     final List<Integer> ls = List.of(10,20,30,40,50,60);
18
19     // Note that the name of the method in Filter is never mentioned here
20     System.out.println(filterAll(ls,x -> x>20)); // [30,40,50,60]
21     System.out.println(filterAll(ls,x -> x>20 && x<60)); // [30,40,50]
22     System.out.println(filterAll(ls,x -> x%20==0)); // [20,40,60]
23   }
24 }
```

Esempio: comandi "programmati" via Runnable

```
1  public class RunnableUtility {
2
3    private static void iterate(final int howMany, final Runnable r){
4      for (int i = 0; i < howMany; i++){
5        r.run();
6      }
7    }
8
9    private static void batchExecution(final List<Runnable> list){
10     for (final Runnable r: list){
11       r.run();
12     }
13   }
14
15   public static void main(String[] args) {
16     iterate(10, ()-> System.out.println("ok"));
17     final List<Runnable> list = List.of(
18         ()->System.out.println("a"),
19         ()->System.out.println("b"),
20         ()->System.out.println("c"),
21         ()->System.exit(0)
22     ); // Inferenza su List.of automatica!
23     batchExecution(list);
24   }
25 }
```

Motivazioni e vantaggi nell'uso delle lambda in Java

Elementi di programmazione funzionale in Java

- Le lambda consentono di aggiungere certe funzionalità della programmazione funzionale in Java, creando quindi una contaminazione OO + funzionale
- Il principale uso è quello che concerne la creazione di funzionalità (metodi) ad alto riuso – ad esempio `filterAll`
- Tali metodi possono prendere in ingresso funzioni, passate con sintassi semplificata rispetto a quella delle classi anonime, rendendo più "naturale" e agevole l'uso di questo meccanismo

Miglioramento alle API di Java

- Concetto di Stream e sue manipolazioni, per lavorare su dati sequenziali (collezioni, file,..)
- Facilitare la costruzioni di software "parallelo" (multicore)
- Supporto più diretto ad alcuni pattern: Command, Strategy, Observer
- Alcune migliorie "varie" nelle API

Interfacce Iterator e Iterable complete

```java
public interface Iterable<T> {

    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

```java
public interface Iterator<E> {

    boolean hasNext();

    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

Uso delle "nuove" interfacce Iterator e Iterable

```java
public class UseIterators {

  // Uso ancor più compatto del for-each, e sue varianti..
  public static void main(String[] args) {
    final List<Integer> list = List.of(10,20,30,40,50,60,70);
    list.forEach(x -> System.out.print(x+" "));
    System.out.println();

    final Collection<Integer> coll = new HashSet<>();
    list.forEach(x -> coll.add(x)); // list.forEach(coll::add);
    System.out.println(coll);

    final Iterator<Integer> it = list.iterator();
    System.out.println(it.next()+" "+it.next());
    it.forEachRemaining( x -> System.out.print(x+" "));
    System.out.println();

    // Nota: Iterable è una interfaccia funzionale..
    final Iterable<Integer> iterable = () -> list.iterator();
    for (final int i: iterable){ // One of my last uses of "for" :)
      System.out.print(i+" ");
    }
    for (final int i: (Iterable<Integer>)()->list.iterator()){
      System.out.print(i+" ");
    }
    System.out.println();

  }
```

Interfaccia java.util.Map – metodi aggiuntivi

```
1  public interface Map<K,V> {
2      ...
3
4      default V getOrDefault(Object key, V defaultValue) {..}
5
6      default void forEach(BiConsumer<? super K, ? super V> action) {..}
7
8      default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {..}
9
10     default V putIfAbsent(K key, V value) {..}
11
12     default boolean remove(Object key, Object value) {..}
13
14     default boolean replace(K key, V oldValue, V newValue) {..}
15
16     default V replace(K key, V value) {..}
17
18     default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
         {..}
19
20     default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
         remappingFunction) {..}
21
22     default V compute(K key, BiFunction<? super K, ? super V, ? extends V>
         remappingFunction) {..}
23
24     default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
         remappingFunction) {..}
25 }
```

Uso delle "nuova" interfacce Map

```java
public class UseMap {

  public static void main(String[] args) {
    final Map<Integer,String> map = new HashMap<>();
    map.put(10, "a");
    map.put(20, "bb");
    map.put(30, "ccc");

    map.forEach((k, v) -> System.out.println(k+" "+v));

    map.replaceAll((k, v) -> v + k); // nuovi valori
    System.out.println(map);
    // {20=bb20, 10=a10, 30=ccc30}

    map.merge(5, ".", String::concat);
    map.merge(10, ".", String::concat);
    System.out.println(map);
    // {20=bb20, 5=., 10=a10., 30=ccc30}

    System.out.println(map.getOrDefault(5, "no")); // "."
    System.out.println(map.getOrDefault(6, "no")); // "no"
  }
}
```

La classe `Optional`

Il problema del `NullPointerException`

- è una eccezione particolarmente annosa (è ora la più frequente)
- a volte è inevitable inizializzare a `null` campi/variabili, o tornare valori null.. ma poi si rischia di ritrovarsi l'eccezione in punti non aspettati
- nella programmazione moderna, gestire l'assenza di una informazione con `null` è inappropriato!

Idea

- la classe `Optional<T>` và usata ove ci si attende opzionalmente un oggetto di tipo `T`
- un oggetto di `Optional<T>` è un wrapper di un `T`, ma potrebbe non contenere nulla, ossia è una collezione di 0 o 1 elemento di tipo `T`
- accedendovi con metodi quali `ifPresent()` o `orElse()` si bypassa il problema del `null`
- c'è comunque un metodo `get()` che rilancia l'eccezione uncheked `NoSuchElementException`
- purtroppo `Optional<T>` NON è serializzabile!

Classe java.util.Optional

```
1  package java.util;
2
3  public final class Optional<T> {
4
5      // metodi statici di costruzione
6      public static<T> Optional<T> empty() {..}
7      public static <T> Optional<T> of(T value) {..}
8      public static <T> Optional<T> ofNullable(T value) {..}
9
10     // selettori
11     public T get() {..} // throws NoSuchElementException on null
12     public boolean isPresent() {..}
13     public T orElse(T other) { .. }
14
15     // uso di funzioni
16     public T orElseGet(Supplier<? extends T> other) {..}
17     public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
         throws X {..}
18     public void ifPresent(Consumer<? super T> consumer) {..}
19     public Optional<T> filter(Predicate<? super T> predicate) {..}
20     public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {..}
21     public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {..}
22
23 }
```

UseOptional

```java
public class UseOptional {

  public static void main(String[] args) {

    final String s = Math.random() > 0.5 ? "high value" : null;
    //Optional<String> opt = Optional.ofNullable(s);
    Optional<String> opt = s == null ? Optional.empty() : Optional.of(s);

    System.out.println("present: "+opt.isPresent());
    System.out.println("orElse: "+opt.orElse("it is actually null"));
    try{
      System.out.println("get: "+opt.get());
    } catch (Exception e){
      System.out.println("get failed.."+e.getClass());
    }

    System.out.print("ifPresent..");
    opt.ifPresent(System.out::println);

  }

}
```

UseOptional2

```java
public class UseOptional2 {

  public static void main(String[] args) throws Exception {

    final Optional<Integer> opt = Optional.of(10);
    final Optional<Integer> opt2 = Optional.empty();
    System.out.println(opt.isPresent());
    System.out.println(opt2.isPresent());

    if (opt.isPresent()){
      System.out.println(opt.get()); // 10
    } else {
      System.exit(0);
    }
    opt.ifPresent(System.out::println); // 10
    opt2.ifPresent(System.out::println); // nothing

    opt.map(x -> x+1).ifPresent(System.out::println); // 11
    opt.map(x -> x+"1").ifPresent(System.out::println); // "101"

    // Your first monad example!
    Optional<Integer> op3 = opt.flatMap(x ->
                opt2.flatMap(y ->
                opt2.flatMap(z -> Optional.of(x+y+z))));
    System.out.println(op3);

    System.out.println(opt.orElseThrow(()->new Exception())); // 10
    System.out.println(opt2.orElseThrow(()->new Exception())); // exc
```

UseOptional3

```java
public class UseOptional3 {

  public static void main(String[] args) {

    final List<Integer> list = Arrays.asList(10,20,30,null,null,50,60,null);
    System.out.println(list);

    final List<Optional<Integer>> olist = new ArrayList<>();
    list.forEach(i -> olist.add(Optional.ofNullable(i)));
    System.out.println(olist);

    final List<Integer> l2 = new ArrayList<>();
    olist.forEach(o -> o.ifPresent(l2::add));
    System.out.println(l2);


    final List<Optional<Integer>> olist2 = new ArrayList<>();
    olist.forEach(o -> olist2.add(o.filter(i -> i<60)));
    System.out.println(olist2);

    final List<Optional<String>> olist3 = new ArrayList<>();
    olist.forEach(o -> olist3.add(o.map(i -> i<40 ? "small" : "big")));
    System.out.println(olist3);
  }

}
```

`Optional` per campi opzionali, e manipolazioni con `map`

Come evitare il `NullPointerException` in una applicazione?

- Non si menzioni mai il `null` nell'applicazione
- Non si lascino variabili o campi non inizializzati
- Si usi `Optional` per campi con contenuto opzionale, inizializzati a `Optional.empty`
- I valori di `Optional` vengano manipolati in modo "dichiarativo" con i suoi metodi, ad esempio, `map`

$\Rightarrow$ Il codice risultante sarà molto più espressivo

Inconveniente:

- e se non si può controllare che il caller di una classe non passi dei `null`?
- si intercettino gli eventuali `null` in ingresso ad ogni metodo, ponendovi un rimedio che non sia il lancio di una eccezione unchecked
- si usi `Objects.requireNonNull()`

Person

```java
public class Person {

  private final String name;
  private Optional<Person> partner = Optional.empty();

  public Person(final String name) {
    this.name = Objects.requireNonNull(name);
  }

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

  public Optional<Person> getPartner() {
    return this.partner;
  }

  public void setPartner(final Person p){
    this.partner = Optional.of(Objects.requireNonNull(p));
  }

  public void removePartner(){
    this.partner = Optional.empty();
  }

  public Optional<String> getPartnerName(){
    return this.partner.map(Person::getName);
  }
}
```

UsePerson

```
1  public class UsePerson {
2
3    public static void main(String[] args) {
4
5      final Person p1 = new Person("Mario");
6      final Person p2 = new Person("Gino");
7      p1.setPartner(p2);
8
9      System.out.println(p1.getName()+" "+
10                        p1.getPartner().isPresent()+" "+
11                        p1.getPartnerName());
12     // Mario true Optional[Gino]
13
14     System.out.println(p2.getName()+" "+
15                           p2.getPartner().isPresent()+" "+
16                           p2.getPartnerName());
17     // Gino false Optional.empty
18   }
19
20 }
```

Outline

Switching

Istruzione `switch`

- un classico costrutto di programmazione strutturata
- corrisponde ad una cascata di `if`, su condizioni di uguaglianza
- selezionato un caso, l'esecuzione riparte da lì, quindi spesso si usa il `break`
- usata tipicamente su `enum` o tipi numerici
- su oggetti, usa `equals`
- il caso `default` usabile come sorta di `else`, opzionalmente

Espressione `switch`

- una variante dello `switch` statement (come l'operatore ternario è variante dell'`if`)
- sintassi ispirata alle lambda
- ogni caso dello switch rappresenta di fatto un return;
- permette di esprimere comodamente funzioni o espressioni per casi

switch statememts

```java
// Assumiamo: enum WorkDay { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
public class SwitchStatement {

    public static boolean isAGoodDay(WorkDay workday){
        boolean result = false;
        switch (workday){
            case MONDAY:
            case TUESDAY:
            case WEDNESDAY:
                result = false;
                break;
            case THURSDAY:
            case FRIDAY:
                result = true;
        }
        return result;
    }

    public static void printWelcomeMessage(WorkDay workday){
        switch (workday){
            case MONDAY: System.out.println("Welcome from weekend!"); break;
            case FRIDAY: System.out.println("Have a nice weekend!"); break;
            default: System.out.println("Nothing to say");
        }
    }

    public static int fibonacci(int i){
        switch (i) {
            case 0: return 1;
            case 1: return 1;
            default: return fibonacci(i-1) + fibonacci(i-2);
        }
```

switch expressions

```java
public class SwitchExpression {

    public static boolean isAGoodDay(WorkDay workday) {
        return switch (workday) {
            case MONDAY, TUESDAY, WEDNESDAY -> false;
            case THURSDAY, FRIDAY -> true;
        };
    }

    public static void printWelcomeMessage(WorkDay workday){
        System.out.println(switch (workday){ // Applico DRY, con lo switch "interno"
            case MONDAY -> "Welcome from weekend!";
            case FRIDAY -> "Have a nice weekend!";
            default -> "Nothing to say";
        });
    }

    public static int fibonacci(int i) {
        return switch (i) {
            case 0, 1 -> 1;
            default -> fibonacci(i - 1) + fibonacci(i - 2);
        };
    }

    public static void main(String[] args){
        Function<Integer, Optional<Integer>> sqrt = i -> switch(i){
            case 1 -> Optional.of(1);
            case 4 -> Optional.of(2);
            case 9 -> Optional.of(3);
            default -> Optional.empty();
        };
        System.out.println(sqrt.apply(5));
```