

# 17

## Lambda expressions

Mirko Viroli  
[mirko.viroli@unibo.it](mailto:mirko.viroli@unibo.it)

C.D.L. Ingegneria e Scienze Informatiche  
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2025/2026

# 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

- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java
- 4 Records
- 5 Switch expressions

# Programmazione funzionale in Java

## Introdotta da Java 8 (2014)

- Un significativo cambio di direzione nella OOP “mainstream” (ispirato da Scala)
- Principale novità: lambda (ossia uno degli elementi fondamentali dello stile di programmazione funzionale)
- Può impattare significativamente il nostro “stile” di programmazione, rendendolo più elegante e chiaro
- . . . specialmente in congiunzione con l’uso di librerie moderne (`Stream` in Java)

## Concettualizzazione dell’idea nella OOP

- Certi metodi hanno argomenti che più che input del problema sono “configurazioni comportamentali”  
⇒ . . . queste sono meglio catturate da lambda invece che oggetti

## 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

```
1 public class FirstComparable {
2
3     public static void main(String[] args) {
4
5         final List<Person> list = new ArrayList<Person>();
6         list.add(new Person("Mario", 1960, true));
7         list.add(new Person("Gino", 1970, false));
8         list.add(new Person("Rino", 1951, true));
9         System.out.println(list);
10
11     // With anonymous class
12     Collections.sort(list, new Comparator<Person>(){
13         public int compare(Person o1, Person o2) {
14             return o1.getBirth() - o2.getBirth();
15         }
16     });
17     System.out.println(list);
18
19     // With full-syntax lambda
20     Collections.sort(list, (Person o1, Person o2) -> {
21         return o1.getBirth() - o2.getBirth();
22     });
23     System.out.println(list);
24
25     // With shortcut-syntax lambda
26     Collections.sort(list, (o1, o2) -> o2.getBirth() - o1.getBirth());
27     System.out.println(list);
28 }
```

## Preview 2: handler eventi con le lambda

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

## Preview 3: iterazioni “dichiarative” con gli stream

```
1 public class FirstStream {  
2  
3     public static void main(String[] args) {  
4         final List<Integer> list = List.of(10,20,30,40,50,60,70,80,90);  
5  
6         // for each element, pass it to method println of System.out  
7         list.stream().forEach(System.out::println); // "10" \n "20" \n "30"  
8  
9         // Filter and print  
10        list.stream().filter(i->i>60).forEach(i -> System.out.print(i+" "));  
11        System.out.println(""); // "70 80 90 "  
12  
13         // Map and print  
14        list.stream().map(i->i+" ").forEach(System.out::print);  
15        System.out.println(""); // "10 20 .. 70 80 90 "  
16  
17         // Map-reduce and print the resulting string  
18        final String s = list.stream().map(i->i+"|").reduce((x,y) -> x+y).get();  
19        System.out.println(s); // "10|20|..|70|80|90|"  
20  
21    }  
22  
23 }
```

# Outline

1 Introduzione

2 Lambda expressions

3 Lambda expressions nell'API di Java

4 Records

5 Switch expressions

# Elementi delle lambda expression

## Che cos'è una lambda

- è una funzione (anonima) con accesso ad uno scope locale
- è applicabile a certi input (0, 1 o più), e dà un risultato (oppure `void`)
- è una “chiusura”, ossia 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
- si usa spesso per “passare del codice” (come lo Strategy)

## Caratteristica specifica di Java

- come vedremo, una lambda è di fatto come un oggetto, 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 più usata, senza typing dei parametri e con expression-body

- $(x_1, \dots, x_n) \rightarrow <\text{exp}>$  // con n argomenti
- $() \rightarrow <\text{exp}>$  // caso speciale a zero argomenti
- $x \rightarrow <\text{exp}>$  // caso special a 1 argomento

Alternative con typing aggiuntivo e/o body

- $(T_1\ x_1, \dots, T_n\ x_n) \rightarrow \{<\text{body}>\}$  // sintassi completa
- $(x_1, \dots, x_n) \rightarrow \{<\text{body}>\}$  // completa non tipata
- $(T_1\ x_1, \dots, T_n\ x_n) \rightarrow <\text{exp}>$  // tipata con body

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

```
1 public class AllLambdas {  
2  
3     private static int mycompare(final String a, final String b) {  
4         return a.compareTo(b);  
5     }  
6  
7     public static void main(String[] args) {  
8         final List<String> list = List.of("a", "bb", "c", "ddd");  
9  
10    // sintassi completa  
11    Collections.sort(list, (String a, String b) -> {  
12        return a.length() - b.length();  
13    });  
14  
15    // sintassi con type inference  
16    Collections.sort(list, (a, b) -> {  
17        return a.length() - b.length();  
18    });  
19  
20    // sintassi con type inference e single-expression body  
21    Collections.sort(list, (a, b) -> a.length() - b.length());  
22  
23    // il body è una chiamata di metodo diretta: userò un method reference  
24    Collections.sort(list, (a, b) -> mycompare(a, b));  
25    Collections.sort(list, AllLambdas::mycompare); // method reference  
26 }  
27 }
```

# I method reference

## Sintassi possibili

1. <class>::<static-method>
  - ▶ sta per (x<sub>1</sub>, ..., x<sub>n</sub>) -> <class>. <static-method>(x<sub>1</sub>, ..., x<sub>n</sub>)
2. <obj>::<method>
  - ▶ sta per (x<sub>1</sub>, ..., x<sub>n</sub>) -> <obj>. <method>(x<sub>1</sub>, ..., x<sub>n</sub>)
3. <class>::<instance-method>
  - ▶ sta per (x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>) -> x<sub>1</sub>. <instance-method>(x<sub>2</sub>, ..., x<sub>n</sub>)
4. <class>::new
  - ▶ sta per (x<sub>1</sub>, ..., x<sub>n</sub>) -> new <class>(x<sub>1</sub>, ..., x<sub>n</sub>)

## 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)

```
1 public class MethodReferences {  
2  
3     private static int staticMyCompare(final String a, final String b) {  
4         return a.compareTo(b);  
5     }  
6     private int instanceMyCompare(final String a, final String b) {  
7         return b.compareTo(a);  
8     }  
9     public static void main(String[] args) {  
10         final List<String> list = List.of("a", "bb", "c", "ddd");  
11         final MethodReferences objAL = new MethodReferences();  
12  
13         Collections.sort(list, (x,y) -> staticMyCompare(x,y));  
14         // <class>::<static-method>  
15         Collections.sort(list, MethodReferences::staticMyCompare);  
16  
17         Collections.sort(list, (x,y) -> objAL.instanceMyCompare(x, y));  
18         // <object>::<instance-method>  
19         Collections.sort(list, objAL::instanceMyCompare);  
20  
21         Collections.sort(list, (x,y) -> x.compareTo(y));  
22         // <class>::<instance-method>  
23         Collections.sort(list, String::compareTo); // same as above  
24     }  
25 }
```

# 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

```
1 public class FirstComparable2 {  
2  
3     public static void main(String[] args) {  
4  
5         final List<Person> list = new ArrayList<Person>();  
6         list.add(new Person("Mario", 1960, true));  
7         list.add(new Person("Gino", 1970, false));  
8         list.add(new Person("Rino", 1951, true));  
9         System.out.println(list);  
10  
11     // Sorting with a lambda  
12     Collections.sort(list, (p1,p2) -> p2.getBirth()-p1.getBirth());  
13     System.out.println(list);  
14  
15     // Nota che sort richiede un Comparator<Person>, che ha il solo metodo:  
16     //     int compare(Persona p1, Persona p2)  
17     // Quindi il codice equivalente generato da javac è:  
18     Collections.sort(list, new Comparator<Person>(){  
19         public int compare(Person p1, Person p2) {  
20             return p2.getBirth()-p1.getBirth();  
21         }  
22     });  
23  
24     System.out.println(list);  
25 }  
26  
27 }
```

# Esempio: funzione riusabile di filtraggio

```
1 public interface Filter<X> {  
2  
3     // Does element x pass the filter?  
4     boolean applyFilter(X x);  
5  
6 }
```

```
1 public class FilterUtility {  
2  
3     public static <X> Set<X> filterAll(Collection<X> set, Filter<X> filter){  
4         final Set<X> newSet = new HashSet<>();  
5         for (final X x: set){  
6             if (filter.applyFilter(x)){  
7                 newSet.add(x);  
8             }  
9         }  
10        return newSet;  
11    }  
12  
13    public static void main(String[] args) {  
14        final List<Integer> ls = List.of(10,20,30,40,50,60);  
15  
16        // Nota che il nome del metodo in Filter non è mai menzionato qui  
17        System.out.println(filterAll(ls,x -> x>20)); // [30,40,50,60]  
18        System.out.println(filterAll(ls,x -> x>20 && x<60)); // [30,40,50]  
19        System.out.println(filterAll(ls,x -> x%20==0)); // [20,40,60]  
20    }  
21 }
```

# Lambda che accedono al loro scope (chiusure)

```
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 pattern template method solo con interfacce

## Esempi di interfacce con metodi di default

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

# Esempio SimpleIterator

```
1 public interface SimpleIterator<X> {  
2  
3     X getNext();  
4  
5     // a template method: i.e., it calls abstract getNext()  
6     default List<X> getListOfNext(int n){  
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 }
```

```
1 public class UseSimpleIterator {  
2  
3     public static void main(String[] args){  
4         final Counter c = new Counter();  
5         final SimpleIterator<Integer> si =  
6             () -> {c.inc(); return c.getValue();};  
7  
8         System.out.println(si.getNext());  
9         System.out.println(si.getListOfNext(20));  
10    }  
11  
12 }
```

# 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

- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java
- 4 Records
- 5 Switch expressions

# 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, e altamente generiche
- 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`, usato nella concorrenza

## 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 }
```

# Esempio: Supplier (e constructor reference)

```
1 public class SupplierExample {  
2  
3     private void check(boolean condition, Supplier<RuntimeException> exception){  
4         if (!condition){  
5             throw exception.get();  
6         }  
7     }  
8  
9     public int access(int[] array, int index){  
10        check(array != null, NullPointerException::new); //method/constructor reference  
11        check(index >=0 && index < array.length,  
12            () -> new IndexOutOfBoundsException("Error at position "+index));  
13        return array[index];  
14    }  
15}  
16}
```

# 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 costruzione di software “parallelo” (multicore)
- Supporto più diretto ad alcuni pattern: Command, Strategy, Observer
- Alcune migliorie “varie” nelle API

# Interfacce Iterator e Iterable complete

```
1 public interface Iterable<T> {
2
3     Iterator<T> iterator();
4
5     default void forEach(Consumer<? super T> action) {
6         Objects.requireNonNull(action);
7         for (T t : this) {
8             action.accept(t);
9         }
10    }
11
12    default Spliterator<T> spliterator() {
13        return Spliterators.spliteratorUnknownSize(iterator(), 0);
14    }
15 }
```

```
1 public interface Iterator<E> {
2
3     boolean hasNext();
4
5     E next();
6
7     default void remove() {
8         throw new UnsupportedOperationException("remove");
9     }
10
11    default void forEachRemaining(Consumer<? super E> action) {
12        Objects.requireNonNull(action);
13        while (hasNext())
14            action.accept(next());
15    }
16 }
```



# Uso delle “nuove” interfacce Iterator e Iterable

```
1 public class UseIterators {  
2  
3     // Uso ancor più compatto del for-each, e sue varianti..  
4     public static void main(String[] args) {  
5         final List<Integer> list = List.of(10,20,30,40,50,60,70);  
6         list.forEach(x -> System.out.print(x+" "));  
7         System.out.println();  
8  
9         final Collection<Integer> coll = new HashSet<>();  
10        list.forEach(x -> coll.add(x)); // list.forEach(coll::add);  
11        System.out.println(coll);  
12  
13        final Iterator<Integer> it = list.iterator();  
14        System.out.println(it.next()+" "+it.next());  
15        it.forEachRemaining( x -> System.out.print(x+" "));  
16        System.out.println();  
17  
18        // Nota: Iterable è una interfaccia funzionale..  
19        final Iterable<Integer> iterable = () -> list.iterator();  
20        for (final int i: iterable){ // One of my last uses of "for" :)  
21            System.out.print(i+" ");  
22        }  
23        for (final int i: (Iterable<Integer>)() -> list.iterator()){  
24            System.out.print(i+" ");  
25        }  
26        System.out.println();  
27    }  
28}
```

# 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
21    default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
22        remappingFunction) {...}
23
24    default V compute(K key, BiFunction<? super K, ? super V, ? extends V>
25        remappingFunction) {...}
26
27    default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
28        remappingFunction) {...}
29 }
```

# Uso delle “nuova” interfacce Map

```
1 public class UseMap {  
2  
3     public static void main(String[] args) {  
4         final Map<Integer, String> map = new HashMap<>();  
5         map.put(10, "a");  
6         map.put(20, "bb");  
7         map.put(30, "ccc");  
8  
9         map.forEach((k, v) -> System.out.println(k + " " + v));  
10  
11        map.replaceAll((k, v) -> v + k); // nuovi valori  
12        System.out.println(map);  
13        // {20=bb20, 10=a10, 30=ccc30}  
14  
15        map.merge(5, ".", String::concat);  
16        map.merge(10, ".", String::concat);  
17        System.out.println(map);  
18        // {20=bb20, 5=., 10=a10., 30=ccc30}  
19  
20        System.out.println(map.getOrDefault(5, "no")); // .  
21        System.out.println(map.getOrDefault(6, "no")); // no  
22    }  
23}
```

# La classe Optional

## Il problema del NullPointerException

- è una eccezione particolarmente annosa (è ora la più frequente)
- a volte è inevitabile 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 considerato inappropriato: il `null` non andrebbe proprio più usato.

## Idea

- la classe `Optional<T>` va 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 unchecked `NoSuchElementException`
- `Optional<T>` NON è serializzabile: serve quindi serializzazione ad-hoc!

# 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)
18        throws X {...}
19    public void ifPresent(Consumer<? super T> consumer) {...}
20    public Optional<T> filter(Predicate<? super T> predicate) {...}
21    public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {...}
22    public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {...}
23}
```

# UseOptional

```
1 public class UseOptional {  
2  
3     public static void main(String[] args) {  
4  
5         final String s = Math.random() > 0.5 ? "high value" : null;  
6         //Optional<String> opt = Optional.ofNullable(s);  
7         Optional<String> opt = s == null ? Optional.empty() : Optional.of(s);  
8  
9         System.out.println("present: "+opt.isPresent());  
10        System.out.println("orElse: "+opt.orElse("it is actually null"));  
11        try{  
12            System.out.println("get: "+opt.get());  
13        } catch (Exception e){  
14            System.out.println("get failed.." +e.getClass());  
15        }  
16  
17        System.out.print("ifPresent..");  
18        opt.ifPresent(System.out::println);  
19  
20    }  
21  
22 }
```

## UseOptional2

```
1 public class UseOptional2 {  
2  
3     public static void main(String[] args) throws Exception {  
4  
5         final Optional<Integer> opt = Optional.of(10);  
6         final Optional<Integer> opt2 = Optional.empty();  
7         System.out.println(opt.isPresent());  
8         System.out.println(opt2.isPresent());  
9  
10        if (opt.isPresent()) {  
11            System.out.println(opt.get()); // 10  
12        } else {  
13            System.exit(0);  
14        }  
15        opt.ifPresent(System.out::println); // 10  
16        opt2.ifPresent(System.out::println); // nothing  
17  
18        opt.map(x -> x+1).ifPresent(System.out::println); // 11  
19        opt.map(x -> x+"1").ifPresent(System.out::println); // "101"  
20  
21        // Your first monad example!  
22        Optional<Integer> op3 = opt.flatMap(x ->  
23                opt2.flatMap(y ->  
24                    opt2.flatMap(z -> Optional.of(x+y+z))));  
25        System.out.println(op3);  
26  
27        System.out.println(opt.orElseThrow(() -> new Exception())); // 10  
28        System.out.println(opt2.orElseThrow(() -> new Exception())); // exc
```

## UseOptional3

```
1 public class UseOptional3 {  
2  
3     public static void main(String[] args) {  
4  
5         final List<Integer> list = Arrays.asList(10, 20, 30, null, null, 50, 60, null);  
6         System.out.println(list);  
7  
8         final List<Optional<Integer>> olist = new ArrayList<>();  
9         list.forEach(i -> olist.add(Optional.ofNullable(i)));  
10        System.out.println(olist);  
11  
12        final List<Integer> l2 = new ArrayList<>();  
13        olist.forEach(o -> o.ifPresent(l2::add));  
14        System.out.println(l2);  
15  
16  
17        final List<Optional<Integer>> olist2 = new ArrayList<>();  
18        olist.forEach(o -> olist2.add(o.filter(i -> i < 60)));  
19        System.out.println(olist2);  
20  
21        final List<Optional<String>> olist3 = new ArrayList<>();  
22        olist.forEach(o -> olist3.add(o.map(i -> i < 40 ? "small" : "big")));  
23        System.out.println(olist3);  
24    }  
25}  
26}
```

# 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`
- ⇒ 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

```
1 public class Person {  
2  
3     private final String name;  
4     private Optional<Person> partner = Optional.empty();  
5  
6     public Person(final String name) {  
7         this.name = Objects.requireNonNull(name);  
8     }  
9  
10    public String getName() {  
11        return this.name;  
12    }  
13  
14    public Optional<Person> getPartner() {  
15        return this.partner;  
16    }  
17  
18    public void setPartner(final Person p){  
19        this.partner = Optional.of(Objects.requireNonNull(p));  
20    }  
21  
22    public void removePartner(){  
23        this.partner = Optional.empty();  
24    }  
25  
26    public Optional<String> getPartnerName(){  
27        return this.partner.map(Person::getName);  
28    }  
29}
```

# 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

1 Introduzione

2 Lambda expressions

3 Lambda expressions nell'API di Java

4 Records

5 Switch expressions

# Records

## Classi implementate come record

- un meccanismo introdotto in Java 14
- serve a dichiarare una classe con certi campi immutabili, e evitare tutto il boilerplate code di solito necessario
- promuove l'immutabilità dei dati, via sintassi concisa: da usare quando possibile!

## Meccanismo:

```
record C(T1 x1, ..., Tn xn) implements I1,...,Ik{...}
```

- realizzata come “`class C implements I1,...,Ik {...}`”, e...
- con campi privati e finali `T1 x1, ..., Tn xn`
- con costruttore con parametri `x1...xn` – costruttore ridefinibile
- con getters `public T1 x1() ... public Tn xn()`
- con `equals`, `hashcode` e `toString`: usano tutti i campi
- non ammette ereditarietà, non ammette aggiunte di campi istanza
- si possono aggiungere altre cose: metodi, membri statici, eccetera

## record example

```
1 import java.util.Objects;
2
3 public class TryRecords {
4
5     // basic record
6     record Person(String name, String surname, int year){}
7
8     interface Product{
9         int id();
10        String name();
11    }
12
13     // a record implementing an interface
14     record ProductImpl(String name, int id) implements Product {
15         public ProductImpl{ // changing primary constructor
16             Objects.requireNonNull(name);
17         }
18         public ProductImpl(int id){ // adding another constructor
19             this("", id);
20         }
21     }
22
23     public static void main(String[] args){
24         Person person = new Person("mario", "rossi", 1980);
25         System.out.println(person + " " + person.name() + " " + person.surname());
26         Product product = new ProductImpl("fruit", 100);
27         System.out.println(product);
28         System.out.println(new ProductImpl(0));
29         // System.out.println(new ProductImpl(null, 101)); // eccezione
30     }
31 }
```

# Outline

- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java
- 4 Records
- 5 Switch expressions

# 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`, optionalmente

## 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 statements

```
1 // Assumiamo: enum WorkDay { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY }
2 public class SwitchStatement {
3
4     public static boolean isAGoodDay(WorkDay workday){
5         boolean result = false;
6         switch (workday){
7             case MONDAY:
8             case TUESDAY:
9             case WEDNESDAY:
10                result = false;
11                break;
12             case THURSDAY:
13             case FRIDAY:
14                 result = true;
15         }
16         return result;
17     }
18
19     public static void printWelcomeMessage(WorkDay workday){
20         switch (workday){
21             case MONDAY: System.out.println("Welcome from weekend!"); break;
22             case FRIDAY: System.out.println("Have a nice weekend!"); break;
23             default: System.out.println("Nothing to say");
24         }
25     }
26
27     public static int fibonacci(int i){
28         switch (i) {
29             case 0: return 1;
30             case 1: return 1;
31             default: return fibonacci(i-1) + fibonacci(i-2);
32         }
33     }
34 }
```

# switch expressions

```
1 public class SwitchExpression {  
2  
3     public static boolean isAGoodDay(WorkDay workday) {  
4         return switch (workday) {  
5             case MONDAY, TUESDAY, WEDNESDAY -> false;  
6             case THURSDAY, FRIDAY -> true;  
7         };  
8     }  
9  
10    public static void printWelcomeMessage(WorkDay workday){  
11        System.out.println(switch (workday){ // Applico DRY, con lo switch "interno"  
12            case MONDAY -> "Welcome from weekend!";  
13            case FRIDAY -> "Have a nice weekend!";  
14            default -> "Nothing to say";  
15        });  
16    }  
17  
18    public static int fibonacci(int i) {  
19        return switch (i) {  
20            case 0, 1 -> 1;  
21            default -> fibonacci(i - 1) + fibonacci(i - 2);  
22        };  
23    }  
24  
25    public static void main(String[] args){  
26        Function<Integer, Optional<Integer>> sqrt = i -> switch(i){  
27            case 1 -> Optional.of(1);  
28            case 4 -> Optional.of(2);  
29            case 9 -> Optional.of(3);  
30            default -> Optional.empty();  
31        };  
32        System.out.println(sqrt.apply(5));  
33    }  
34}
```

# switch patterns

```
1 public class SwitchPatterns {  
2  
3     record Person(String name, String surname, int year){}  
4  
5     public static String asString(Object obj) {  
6         return switch (obj) {  
7             case null -> "null";  
8             case String str -> str;  
9             case Integer i when i>=0 -> "Positive: "+i;  
10            case Integer i -> "Negative: "+i;  
11            case Person(var n, var s, var y) -> "(" + n + "," + s + "," + y + ")";  
12            default -> obj.toString();  
13        };  
14    }  
15  
16    public static void main(String[] args){  
17        System.out.println(asString(null));  
18        System.out.println(asString("ciao"));  
19        System.out.println(asString(1));  
20        System.out.println(asString(-1));  
21        System.out.println(asString(new Person("mario", "rossi", 1900)));  
22        System.out.println(true);  
23    }  
24}
```