

# 16

## Java 8: Lambda expressions

Mirko Viroli  
`mirko.viroli@unibo.it`

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

a.a. 2018/2019



## Goal della lezione

- Illustrare il concetto di lambda
- Dettagliare il supporto alle lambda in Java 8

## Argomenti

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

- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java



# Le novità di Java 8

## Una release molto attesa, che “rincorre” C# e Scala

- Molte funzionalità rimandate da Java 7
- Java 8 disponibile dall'estate 2014
- 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
- Impatta varie librerie

## 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<Persona> list = new ArrayList<Persona>();
6         list.add(new Persona("Mario",1960,true));
7         list.add(new Persona("Gino",1970,false));
8         list.add(new Persona("Rino",1951,true));
9         System.out.println(list);
10
11         // Without lambdas
12         Collections.sort(list, new Comparator<Persona>(){
13
14             public int compare(Persona o1, Persona o2) {
15                 return o1.getAnno() - o2.getAnno();
16             }
17
18         });
19         System.out.println(list);
20
21         // With lambdas
22         Collections.sort(list, (p1,p2) -> p2.getAnno() - p1.getAnno());
23         System.out.println(list);
24     }
25
26 }
```

# 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!!");
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..");
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         b1.addActionListener( e -> System.out.println("Hello!!"));
6
7         final JButton b2 = new JButton("Quit");
8         b2.addActionListener( e -> {
9             System.out.println("Quitting..");
10            System.exit(0);
11        });
12
13        final JFrame frame = new JFrame("Events Example");
14        frame.setSize(320, 200);
15        frame.getContentPane().setLayout(new FlowLayout());
16        frame.getContentPane().add(b1);
17        frame.getContentPane().add(b2);
18        frame.setVisible(true);
19    }
20 }
```



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

```
1 public class FirstStream {
2
3     public static void main(String[] args) {
4         final List<Integer> list = Arrays.asList(10,20,30,40,50,60,70,80,90);
5
6         // just print all
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 }
```





- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java



# 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
- per calcolare il risultato potrebbe usare qualche variabile nello scope in cui è definita
- è usabile come “valore” (quindi, come dato), ossia è passabile a metodi, altre funzioni, o memorizzata in variabili
- ossia si può “passare” del “codice”

## Caratteristica specifica di Java

- come vedremo, il tipo di una lambda è sempre quello di una **interface** detta “funzionale”
- metodi static o istanza possono essere usati a mo' di lambda

# Come si esprime una lambda

## Sintassi possibili

- $(T1\ x1, \dots, Tn\ xn) \rightarrow \{<body>\}$
- $(x1, \dots, xn) \rightarrow \{<body>\}$
- $x \rightarrow \{<body>\}$
- $(T1\ x1, \dots, Tn\ xn) \rightarrow <exp>$
- $(x1, \dots, xn) \rightarrow <exp>$
- $x \rightarrow <exp>$
- .. oppure un “method reference”

## Ossia:

- 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     // A static method
4     private static int mycompare(final String a, final String b) {
5         return a.compareTo(b);
6     }
7
8     public static void main(String[] args) {
9         final List<String> list = Arrays.asList("a", "bb", "c", "ddd");
10
11         Collections.sort(list, (String a, String b) -> {
12             return a.length() - b.length();
13         });
14         System.out.println(list); // [a, c, bb, ddd]
15
16         Collections.sort(list, (a, b) -> {
17             return a.length() - b.length();
18         });
19         System.out.println(list); // [a, c, bb, ddd]
20
21         Collections.sort(list, (String a, String b) -> a.length() - b.length());
22         System.out.println(list); // [a, c, bb, ddd]
23
24         Collections.sort(list, (a, b) -> a.length() - b.length());
25         System.out.println(list); // [a, c, bb, ddd]
26
27         Collections.sort(list, (a, b) -> mycompare(a, b));
28         System.out.println(list); // [a, c, bb, ddd]
```

# Come si esprime una lambda: method references

## Sintassi possibili

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

## Ossia:

- Descrivibile come metodo (statico o non), o costruttore..
- Usabile “naturalmente” (e opzionalmente) quando la lambda non fa altro che chiamare un metodo usando “banalmente” i suoi input

# Esempi di Lambda

```
1 public class AllLambdas2 {
2
3     // A static method
4     private static int mycompare(final String a, final String b) {
5         return a.compareTo(b);
6     }
7     // An instance method
8     private int mycompare2(final String a, final String b) {
9         return b.compareTo(a);
10    }
11    public static void main(String[] args) {
12        final List<String> list = Arrays.asList("a", "bb", "c", "ddd");
13
14        Collections.sort(list, (a, b) -> a.length() - b.length());
15        System.out.println(list); // [a, c, bb, ddd]
16
17        Collections.sort(list, (x,y) -> mycompare(x,y));
18        System.out.println(list); // [a, c, bb, ddd]
19
20        Collections.sort(list, AllLambdas2::mycompare);
21        System.out.println(list); // [a, bb, c, ddd]
22
23        Collections.sort(list, new AllLambdas2()::mycompare2);
24        System.out.println(list); // [ddd, c, bb, a]
25    }
26 }
```

# Dove si può usare una lambda?

## Definizione di interfaccia “funzionale” (def. non definitiva)

- E' 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
- Una nuova istruzione a livello di bytecode evita di costruirsi effettivamente un `.class` per ogni lambda

# Generazione automatica della classe anonima

```
1 public class FirstComparable2 {
2
3     public static void main(String[] args) {
4
5         final List<Persona> list = new ArrayList<Persona>();
6         list.add(new Persona("Mario",1960,true));
7         list.add(new Persona("Gino",1970,false));
8         list.add(new Persona("Rino",1951,true));
9         System.out.println(list);
10
11         // Sorting with a lambda
12         Collections.sort(list, (p1,p2) -> p2.getAnno()-p1.getAnno());
13         System.out.println(list);
14
15         // Nota che sort richiede un Comparator<Persona>, 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<Persona>(){
19             public int compare(Persona p1, Persona p2) {
20                 return p2.getAnno()-p1.getAnno();
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 = Arrays.asList(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

```
1 public class ChangeButton extends JFrame{
2
3     private static final long serialVersionUID = 148347709787601235L;
4
5     private int counter = 0;
6
7     public ChangeButton(){
8         super("Changing button example");
9         final JButton button = new JButton("Val: ");
10        button.addActionListener(
11            e -> button.setText("Val: " + ChangeButton.this.counter++));
12        // button è una variabile finale nel metodo
13        // ChangeButton.this.counter è un campo nella enclosing instance
14
15        this.setSize(320, 200);
16        this.getContentPane().setLayout(new FlowLayout());
17        this.getContentPane().add(button);
18        this.setVisible(true);
19    }
20
21    public static void main(String[] args){
22        new ChangeButton();
23    }
24 }
```

# 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 “behaviour” 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

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

```
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 }
```

## Uso

- da usare opzionalmente per interfacce funzionali
- il compilatore la usa per assicurarsi che l'interfaccia sia funzionale, ossia che vi sia un solo metodo "astratto"
- 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 }
```



- 1 Introduzione
- 2 Lambda expressions
- 3 Lambda expressions nell'API di Java**



# 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

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`





# Package `java.util.function`

## Interface 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 = Arrays.asList(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=Arrays.asList(
18             ()->System.out.println("a"),
19             ()->System.out.println("b"),
20             ()->System.out.println("c"),
21             ()->System.exit(0)
22         ); // Inferenza su asList 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

Realizzare “ogni” algoritmo su “sequenze”  
in una sola istruzione



# “Nuove” interfacce Iterator e Iterable

```
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 = Arrays.asList(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
- gestire l'assenza di una informazione con `null` è inappropriato!

## Idea

- la classe `Optional<T>` v'è 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`
- 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)
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 }
```

