Lesson 3.8 Lambda Expressions and Java 8

Software Analysis and Design

2º Year, Computer Science

Universidad Autónoma de Madrid

Index

- **New concepts in interfaces**
  - **Default methods**
  - **Static methods**
- Lambda expressions
- Exercises
- Conclusions and bibliography

New concepts in interfaces

- Consider the following interface

```
public interface Tree<T>{
  T getElement();
  Tree<T> leftChild();
  Tree<T> rightChild();
  boolean isLeaf();
  boolean isEmpty();
  Tree<T> search(T o);
}
```

- For ease of use, we could give a `default` implementation for some methods
- This implementation can use other methods of the interface
- Classes do not need to give an implementation of a default method, but they can

default methods in interfaces

```
public interface Tree<T>{
  T getElement();
  Tree<T> leftChild();
  Tree<T> rightChild();
  default boolean isLeaf() {
       return  this.leftChild().isEmpty() && this.rightChild().isEmpty();
  }
  boolean isEmpty();
  default Tree<T> search(T o) {
    if (this.getElement().equals(o)) return this;
    else {
       Tree<T> result = null;
       if (! this.leftChild().isEmpty() ) result = this.leftChild().search(o);
       if (result != null) return result;
       if (! this.rightChild().isEmpty() ) result = this.rightChild().search(o);
       if (result != null) return result;
    }
    return null;
  }
}
```

`default` methods: motivation

- Being able to add methods to an interface without breaking already existing code (the new methods of the interface would be default methods)

- Specify methods that are optional
  - Give an implementation that returns an exception

- Facilitate the implementation of interfaces (similar to an abstract class with a reference implementation)

Differences with abstract classes

- An interface does not have internal state
  - It cannot declare attributes (instance variables), but just constants

- A class can only extend one class, while it can implement multiple interfaces

- The purpose of interfaces is still specifying "*what*" (method signatures) and not "*how*" (code in methods)

default methods and multiple inheritance

- Without default methods, there are no name collision problems with multiple inheritance or multiple implementation
- The method has just one code block, implemented in the class

```java
interface Alpha {
  int method1();
}

interface Beta {
  int method1();
}

public class Test implements Alpha, Beta{
  @Override public int method1() { return 42; } // OK!
}
```

default methods and multiple inheritance

- If a default method is implemented in a class, such implementation has preference

```java
interface Alpha {
   default int method1() { return 23;}
}

interface Beta {
   int method1();
}

public class Test implements Alpha, Beta{
  @Override public int method1() { return 42; }

  public static void main(String... args) {
    System.out.println(new Test().method1());
    // the one in the class has preference
  }
}
```

default methods and multiple inheritance

- If there are method name collisions and some of them has code, the class should give an implementation

```
interface Alpha {
   default int method1() { return 23;}
}

interface Beta {
   default int method1() { return 89; }
}

public class Test implements Alpha, Beta{  // Error!
                                           // implementation required
  public static void main(String... args) {
    System.out.println(new Test().method1());
  }
}
```

default methods and multiple inheritance

- If there are method name collisions and some of them has code, the class should give an implementation

```
interface Alpha {
   default int method1() { return 23;}
}

interface Beta {
   default int method1() { return 89; }
}

public class Test implements Alpha, Beta{
  // Use the one in Alpha…
  @Override public int method1() {return Alpha.super.method1(); }
  public static void main(String... args) {
    System.out.println(new Test().method1());
  }
}
```

default methods and multiple inheritance

- The notation `SuperType.super.method()` is usable also in interfaces

```
interface Alpha {
   default int method1() { return 23;}
}

interface Beta extends Alpha {
   default int method1() { return Alpha.super.method1()+1; }
}

public class Test implements Beta{
  public static void main(String... args) {
    System.out.println(new Test().method1());  // 24
  }
}
```

Static methods in interfaces

- It is possible to add static methods to an interface, just like in classes
- Useful to define libraries
  - □ Example: creation of some useful comparators in Comparator<T>

```
public interface Comparator<T> {
   int compare(T o1, T o2);
   static <T extends Comparable<? super T>> Comparator<T> naturalOrder() { … }

   static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) { … }
   static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) { … }
   //…
}
```

Exercise

- Add a static method to the `Tree` interface that returns a `Comparator` that
  - □ Compares the value of the root nodes (if one is empty and the other is not, the bigger is the non-emtpy one)
  - □ If roots are not null, compares by natural order

Index

- New concepts in interfaces
- **Lambda Expressions**
  - **Introduction and examples**
  - **Lambda expressions**
  - **Streams**
- Exercises
- Conclusions and bibliography

Introduction and Examples

What are lambda expressions?

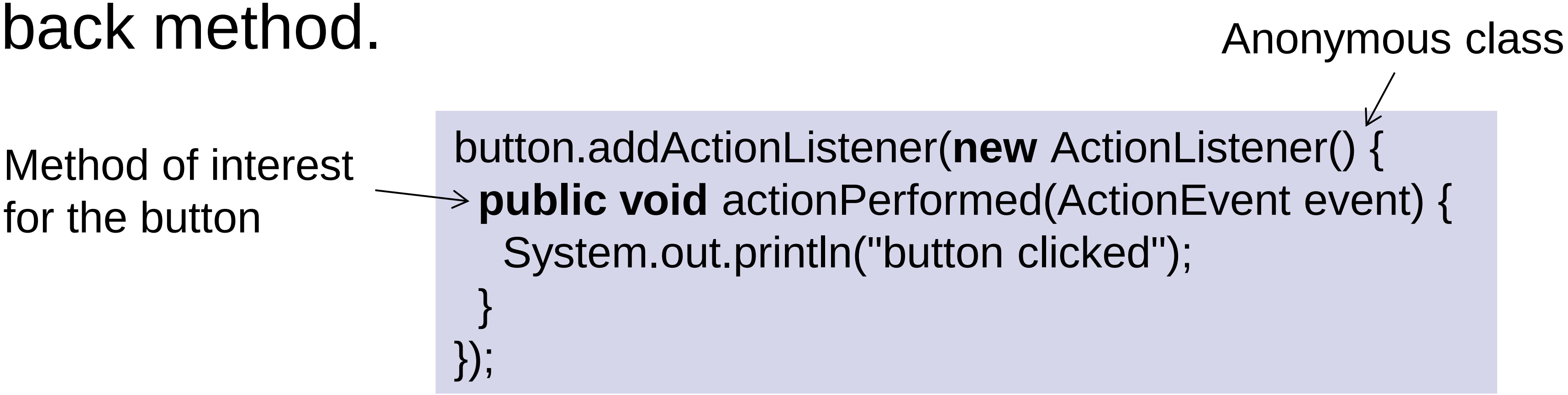
- Functions as first-level concepts
  - Anonymous, without a name
  - We can pass them as parameters

- In Java 8 they are called lambda expressions
- The name comes from λ-calculus (Alonzo Church)
- In other languages (e.g, Ruby) *closures* are a similar concept

- They promote a programming style closer to functional programming
  - Function chaining, operating over *streams*
  - More easily parallelizable (useful to process large amounts of data – Big Data computations)
  - More intentional and less verbose code

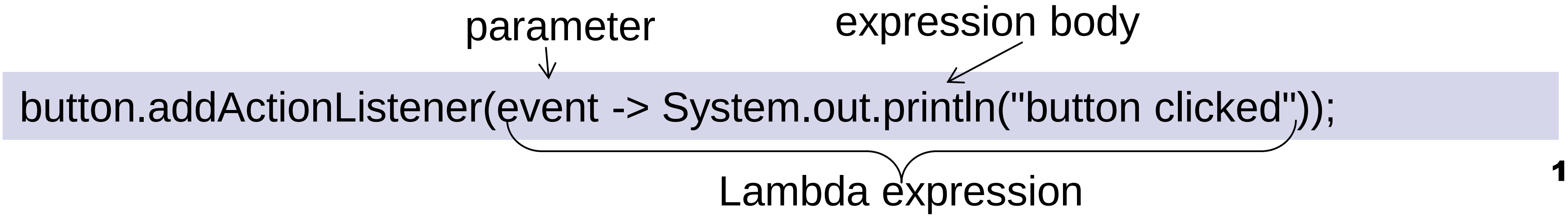
Lambda expressions. Example.

- In Swing, we often need to configure graphical components with *callback* methods, that are executed when an event occurs
- Before Java 8, we needed to define a class to define the callback method.



```
button.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent event) {
   System.out.println("button clicked");
  }
});
```

- A lambda expression permits a more concise syntax



```
button.addActionListener(event -> System.out.println("button clicked"));
```

Another example: filtering a list

Programming using Java 7 style

```
           class Product {
             private int price;
             public Product(int p) { this.price = p; }
             public int getPrice() { return this.price; }
           }


List<Product> products = Arrays.asList(
                  new Product(20),
                  new Product(40),
                  new Product (5));

List<Product> discounts = new ArrayList<Product>();

for (Product p : products)
    if (p.getPrice()>10.0)
      discounts.add(p);
```

Another example: filtering a list

Programming with lambdas

```
           class Product {
             private int price;
             public Product(int p) { this.price = p; }
             public int getPrice() { return this.price; }
           }


List<Product> products = Arrays.asList(
                  new Product(20),
                  new Product(40),
                  new Product (5));

List<Product> discounts = products.stream().
           filter(p -> p.getPrice()>10.0).     // filter those > 10
           collect(Collectors.toList());       // store them in a list
```

Syntactic sugar...

(and some more things)

```
Stream<Product> filter(Predicate<? super Product> a)
```

```
@FunctionalInterface
public interface Predicate<T> {
   //… more things
   boolean test(T t);
}
```

The compiler generates an anonymous class

```
List<Product> descs2 = products.stream().
    filter(new Predicate<Product>() {
                @Override public boolean test(Product a) {
                    return a.getPrice()>10.0;
                }
    }).collect(Collectors.toList());
```

Lambda Expressions

Lambda Expressions

What are they?

- A block of code with no name, made of:
  - List of formal parameters,
  - Separator “->”
  - Body.

```
(int x) -> x + 1
```

- Looks like a method, but it is not: it is an instance of a functional interface

- More precisely, it is a compact notation for an instance of an anonymous class, typed by a functional interface

Lambda expressions

What are they?

- A functional interface is an interface with a unique non-default, non-static method
- Some important functional interfaces

Name	Arguments	Return	Functional method	Examples
Predicate<T>	T	boolean	test(T t)	Does the product has a discount?
Consumer<T>	T	void	accept(T t)	Print a value
Function<T,R>	T	R	apply(T t)	Obtain the price of a Product
Supplier<T>	None	T	get()	Creation of an objet
UnaryOperator<T>	T	T	apply(T t)	Logical negation (!)
BinaryOperator<T>	(T, T)	T	apply(T t, T u)	Multiply two numbers (*)

- Some of them have specializations: IntConsumer
- Others contain *default* and *static* utility methods

Lambda expressions

What are they?

- Lambda expressions do not have
  - Name
  - Declaration of return type (it is inferred)
  - *throws* clause (it is inferred)
  - Declaration of generic types
- The types of the formal parameters can be omitted (implicit vs explicit lambdas)
  - Either all or none omitted
- If the type is included, we can add the `final` modifier to the parameters

Parameters and return

- With zero parameters and no return

```
Runnable noArguments = () -> System.out.println("Hello World");
noArguments.run();
```

```
 /* Equivalent to
 Runnable noArguments = new Runnable() {
                 @Override public void run() {
                      System.out.println("Hello World");
                 }
           };
 */
```

Parameters and return

- The following syntax is incorrect:

❌

```
Runnable noArguments = -> System.out.println("Hello World");
```

Parameters and return

- With one parameter, several instructions and no return:

```
  Consumer<Product> consumer = p -> {
      p.increasePrice(10);
      System.out.println(p.getName()+": "+p.getPrice());
  };


List<Product> products = Arrays.asList( new Product(20, "Salt"),
                                        new Product(40, "Sugar"),
                                        new Product (5, "Wine"));


products.forEach(p -> {
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
});

products.forEach(consumer);  // equivalent to the previous code
```

Parameters and return

- The following syntaxes are equivalent:

✓

```
Consumer<Product> consumer = p -> { // implicit lambda
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

✓

```
Consumer<Product> consumer = (p) -> {
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

✓

```
Consumer<Product> consumer = (Product p) -> {  // explicit lambda
    p.increasePrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

Parameters and return

- The following syntax is incorrect:

```
Consumer<Product> consumer = Product p -> {
    p.incrementPrice(10);
    System.out.println(p.getName()+": "+p.getPrice());
};
```

Parameters and return

- With two parameters and with return:

```
List<Integer> numbers = Arrays.asList(1, 1, 2, 3, 5, 8, 13);

// Optional is a type that admits a result or null…
// …permits a “functional” notation for if…then…else
Optional<Integer> result =
       numbers.stream().
               reduce((x, y) -> x+y); // BinaryOperator<Integer>

System.out.println("Sum="+result.orElse(0));
// if there is no result, prints 0
```

Parameters and return

- The following syntaxes are equivalent:

✓

```
Optional<Integer> result =
   numbers.stream().
          reduce((x, y) -> { return x+y; });
```

✓

```
Optional<Integer> result =
   numbers.stream().
      reduce((Integer x, Integer y) -> { return x+y; });
```

Context variables

- Inside a lambda, we can use external context variables that are final…

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new
Product(40, "Sugar"), new Product (5, "Wine"));

final int increment = 10;

Products.forEach(p -> {
    p.increasePrice(increment);  //increment is final, we can use it
    System.out.println(p.getName()+": "+p.getPrice());
});
```

Context variables

- … o effectively final

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new
Product(40, "Sugar"), new Product (5, "Wine"));

int increment = 10;

Products.forEach(p -> {
    p.increasePrice(increment);  // we do not change increment, OK!
    System.out.println(p.getName()+": "+p.getPrice());
});
```

Context variables

- … o effectively final

```
List<Product> products = Arrays.asList(new Product(20, "Salt"), new
Product(40, "Sugar"), new Product (5, “Wine"));

int increment = 10;

Products.forEach(p -> {
    increment+=3;               // ERROR!
    p.increasePrice(increment); // we do not change increment, OK!
    System.out.println(p.getName()+": "+p.getPrice());
});
```

References to methods

- A reference to a method is an “abbreviation” for a lambda that uses such method
- The method is not called in that moment, it is simply a lambda
- Syntax: `<QualifiedName>::<methodName>`

```
class Product {
  private int price;
  private String name;

  public Product(int p, String n) { this.price = p; this.name = n; }
  public int getPrice() { return this.price; }
  public String getName() { return this.name; }
}
// Obtain the name of the products that start by S,
// without repetition

Set<String> descs2 = products.stream().
               map(Product::getName).   // a reference to a method
               filter( s -> s.startsWith("S")).
               collect(Collectors.toSet());
```

References to methods

```
class Product {
  private int price;
  private String name;

  public Product(int p, String n) { this.price = p; this.name = n; }
  public int getPrice() { return this.price; }
  public String getName() { return this.name; }
}

Set<String> descs2 = products.stream().
               map(p -> p.getName()).  // equivalent 
               filter( s -> s.startsWith("S")).
               collect(Collectors.toSet());
```

References to methods

Types

`<TypeName>::<staticMethod>`	A reference to a static method of a class, interface or enum
`<refObject>::<instanceMethod>`	A reference to an instance method of the object
`<ClassName>::<instanceMethod>`	A reference to an instance method of the class
`<TypeName>.super::<instanceMethod>`	A reference to a method of the superclass of the current object
`<ClassName>::new`	A reference to the class constructor
`<ArrayTypeName>::new`	A reference to the constructor of the array type

References to object methods

```
class Store {
  private List<Product> products = new ArrayList<Product>();

  public Store(Product...products) {
     this.products.addAll(Arrays.asList(products));
  }
  public String getName(Product p) { return p.getName(); }
  public Stream<Product> getProducts() { return this.products.stream(); }
}

class Product { /* as before */}


Store alm = new Store(new Product(20, "Salt"));

Set<String> descs2 = alm.getProducts().
               map(alm::getName). // a reference to an object method
               filter( s -> s.startsWith("S")).
               collect(Collectors.toSet());
```

Exercise

Filters and maps

Ejercise

- Design a class that sequentially stores data of any type, and that can return a filtered sequence of that data by configurable criteria.

Exercise: *currying*

- In functional programming, currying is a widely used technique to reduce the parameters of a function
- Given a function f: X × Y → Z, curry(f) returns a function h: X → (Y → Z).
- That is, h takes an argument of type X, and returns a function Y → Z, defined in such a way that h(x)(y) = f(x, y).
- To do:
  - □ Implement *curry* using lambda expressions.
  - □ Hint: the interface BiFunction<X, Y, Z> can be used to model f, and Function<Y, Z> to model h.

Exercise: *currying*

- In functional p dely used technique to nction
- Given a funct ns a function h: X → (Y → Z)
- That is, h tak returns a function Y → h(x)(y) = f(x, y).
- To do:
  - □ Implemen ons.
  - □ Hint: the i can be used to model f, ar

Haskell Curry

Solution

```
package currying;

import java.util.function.*;

public class Currying {
  private <X, Y, Z> Function<X, Function<Y, Z>> curry(BiFunction<X, Y, Z> f){
    return x -> (y -> f.apply(x, y));
  }

  public static void main(String ...args) {
    Currying c = new Currying();
    Integer result =
      c.<Integer, Integer, Integer>curry((x, y) -> x + y ).
        apply(3).
        apply(4);
     System.out.println(result);
   }
}
```

Ambiguity and casting

```
@FunctionalInterface interface IntegerReduce {
  int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
   String join (String x, String y);
}

public class Joiner {
  public String doJoin (StringReduce sj) { return sj.join("Java", "8");}
  public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

  public static void main(String[] args) {
    Joiner j = new Joiner();
    System.out.println(j.doJoin((x, y) -> x + y));
    // Error:  The method doJoin(StringReduce) is ambiguous  for the type Joiner
  }
}
```

Ambiguity and casting

```
@FunctionalInterface interface IntegerReduce {
  int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
   String join (String x, String y);
}

public class Joiner {
  public String doJoin (StringReduce sj) { return sj.join("Java", "8");}
  public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

  public static void main(String[] args) {
    Joiner j = new Joiner();
    System.out.println(j.doJoin((StringReduce)(x, y) -> x + y));
  }
}
```

Ambiguity and casting

```
@FunctionalInterface interface IntegerReduce {
  int join (int x, int y);
}
@FunctionalInterface interface StringReduce {
   String join (String x, String y);
}

public class Joiner {
  public String doJoin (StringReduce sj) { return sj.join("Java", "8");}
  public int doJoin(IntegerReduce ij) { return ij.join(64, 128); }

  public static void main(String[] args) {
    Joiner j = new Joiner();
    System.out.println(j.doJoin((String x, String y) -> x + y));
    // Equivalent to the previous one
  }
}
```

Functional interfaces

- A functional interface is an interface that has exactly one abstract method.

- Methods not qualifying for the unique method of a functional interface:
  - *default* methods
  - static methods
  - methods inherited from Object

- Functional interfaces can optionally be annotated with *@FunctionalInterface* (in `java.lang`)
  - The compiler checks that the declared interface is functional

Example (1/2)

// An object system with dynamic methods embedded
// in Java

```
@FunctionalInterface interface Method { // To which standard interface
  void exec(ProtoObject o);             // is it equivalent?
}

public class ProtoObject {
  private HashMap<String, Object> slots = new HashMap<>();
  private HashMap<String, Method> methods = new HashMap<>();

  public void add (String name, Method m) { this.methods.put(name, m); }
  public void add (String name, Object v) { this.slots.put(name, v); }
  public Object get (String name) { return this.slots.get(name); }
  public void exec (String name) { this.methods.get(name).exec(this); }
  @Override public String toString() { return this.slots.toString(); }
}
```

Example (2/2)

```java
public class Main {
  public static void main(String[] args) {
    ProtoObject p = new ProtoObject();
    p.add("name", "Leonard Nimoy");
    p.add("age", 83);
    p.add("incrementAge",
           self -> {
                      self.add( "age",
                               ((Integer)self.get("age"))+1);
                   }
    );
    p.add("print",
           self -> {
                      System.out.println("name: "+self.get(“age")+
                      "\n”+“age: "+self.get(“age")+" years.");
                    }
    );
    System.out.println(p);
    p.exec("incrementAge");
    p.exec("print");
    System.out.println(p);
  }
}
```

```
Output:
{name=Leonard Nimoy, age=83}
name: Leonard Nimoy
age: 84 years.
{name=Leonard Nimoy, age=84}
```

Exercise

- Modify the previous example so that:
  - An object (the prototype) can be cloned.
  - The created object stores a reference to its prototype
  - If we access an object variable, and that object has not given it a value, or does not have it, the variable is sought in the prototype.
  - Adding a method or variable in the prototype is reflected in its clones, but not the other way round.

Some languages with similar working scheme:
Self: http://en.wikipedia.org/wiki/Self_%28programming_language%29
JavaScript: http://en.wikipedia.org/wiki/JavaScript

Generic functional interfaces

- A functional interface can have generic parameters.
- Example:

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

Generic functional interfaces

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

  public Person(String n, int e) { this.name = n; this.age = e; }
  public String toString() { return "name: "+this.name+" age: "+this.age; }
  public int getAge() { return this.age; }
}

public class Compare {
  public static void main(String[] args) {
   List<Persona> list = Arrays.asList(new Person("Leonard Simon Nimoy", 83),
                                      new Person("William Shatner", 84),
                                      new Person("Jackson DeForest", 79));

   Collections.sort(list, (x, y) -> x.getAge() - y.getAge());
   System.out.println(list);
   Collections.sort(list, (x, y) -> y.getAge() - x.getAge());
   System.out.println(list);
  }
}
```

Use of Functional Interfaces

Function and its specializations

```
import java.util.function.*;

public class FunctionExample {
  public static void main(String[] args) {
    // Using Function and its specializations
    Function<Integer, Integer> square = x -> x * x;
    IntFunction<String> toStrn = x -> String.valueOf(x);// From int to String
    ToIntFunction<Float> floor   = x -> Math.round(x);// From float to Integer
    UnaryOperator<Integer> square2 = x -> x * x; // From Integer to Integer
    System.out.println(square.apply(5));
    System.out.println(toStrn.apply(5));
    System.out.println(floor.applyAsInt(5f));
    System.out.println(square2.apply(5));
  }
}
```

Output

```
25
5
5
25
```

Function

Some default and static methods

```
@FunctionalInterface
public interface Function<T, R> {
   R apply(T t);   // The functional method
   default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
         Objects.requireNonNull(before);
         return (V v) -> apply(before.apply(v));
   }
  default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
         Objects.requireNonNull(after);
         return (T t) -> after.apply(apply(t));
  }
  static <T> Function <T, T> identity() {
         return t->t;
  }
}
```

Composing Functions

```
public class ComposedFunctions {
  public static void main(String[] args) {
    // Create two functions
    Function<Long, Long> square = x -> x * x;
    Function<Long, Long> addOne = x -> x + 1;
    // Compose functions from the two functions
    Function<Long, Long> squareAddOne = square.andThen(addOne);
    Function<Long, Long> addOneSquare = square.compose(addOne);
    // Get an identity function
    Function<Long, Long> identity = Function.<Long>identity();
    // Test the functions
    long num = 5L;
    System.out.println("Number : " + num);
    System.out.println("Square and then add one: " + squareAddOne.apply(num));
    System.out.println("Add one and then square: " + addOneSquare.apply(num));
    System.out.println("Identity: " + identity.apply(num));
  }
}
```

Output:

```
Number : 5
Square and then add one: 26
Add one and then square: 36
Identity: 5
```

Predicate

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef) ? Objects::isNull  : object -> targetRef.equals(object);
    }
}
```

Predicate

```java
public class Predicates {
    public static void main(String[] args) {
        // Create some predicates
        Predicate<Integer> greaterThanTen = x -> x > 10;
        Predicate<Integer> divisibleByThree = x -> x % 3 == 0;
        Predicate<Integer> divisibleByFive = x -> x % 5 == 0;
        Predicate<Integer> equalToTen = Predicate.isEqual(null);
        // Create predicates using NOT, AND, and OR on other predicates
        Predicate<Integer> lessThanOrEqualToTen=greaterThanTen.negate();
        Predicate<Integer> divisibleByThreeAndFive=divisibleByThree.and(divisibleByFive);
        Predicate<Integer> divisibleByThreeOrFive=divisibleByThree.or(divisibleByFive);
        // Test the predicates
        int num = 10;
        System.out.println("Number: " + num);
        System.out.println("greaterThanTen: " + greaterThanTen.test(num));
        System.out.println("divisibleByThree: " + divisibleByThree.test(num));
        System.out.println("divisibleByFive: " + divisibleByFive.test(num));
        System.out.println("lessThanOrEqualToTen: " + lessThanOrEqualToTen.test(num));
        System.out.println("divisibleByThreeAndFive: " +
                            divisibleByThreeAndFive.test(num));
        System.out.println("divisibleByThreeOrFive: " +
                            divisibleByThreeOrFive.test(num));
        System.out.println("equalsToTen: " + equalToTen.test(num));
    }
}
```

Intersection types and lambdas

- An intersection type (new in Java 8) is an intersection or subtype of multiple types.

- The expression: (Type1 & Type2 & Type3) is a new type, the intersection of the three types.

```
Serializable comp = (Comparator<Person> & Serializable)
                         (x, y) -> x.getAge() - y.getAge();
```

Exercise (1/2)

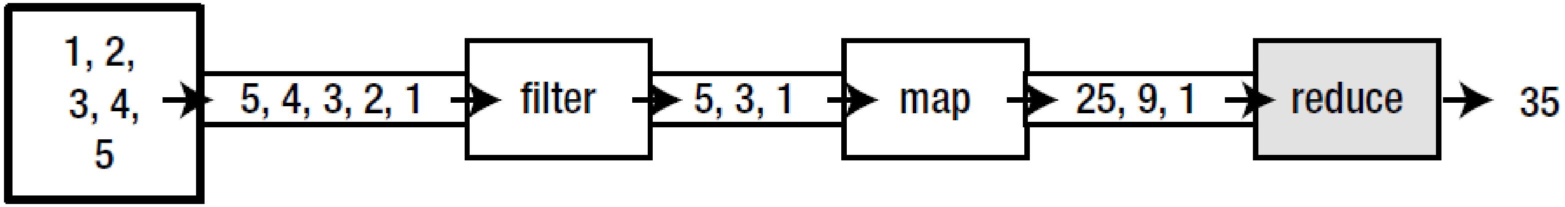
- Using lambdas, create a simulator for state machines.
- A state machine is made of states and a series of variables, which we assume to be of the Integer type.
- The machine transitions from one state to another one when an event (of type String) occurs.
- States can have associated actions, which are executed when exiting the state due to some event, and can for example modify variables.

Exercise (2/2)

```
public class Main {
  public static void main(String[] args) {
    StateMachine sm = new StateMachine("Light", "num"); // name y variable
    State s1 = new State("off");
    State s2 = new State("on");
    s1.addEvent("switch", s2);
    s2.addEvent("switch", s1);
    s1.action((State s, String e) -> s.set("num", s.get("num")+1) );
    s2.action((State s, String e) -> s.set("num", s.get("num")+1) );

    sm.addStates(s1, s2);
    sm.setInitial(s1);

    System.out.println(sm);
    MachineSimulator ms = new MachineSimulator(sm);
    ms.simulate(Arrays.asList("switch", "switch"));
   }
}
```

```
Output:

Machine Light : [off, on]
switch: from [off] to [on]
 Machine variables: {num=1}
switch: from [on] to [off]
 Machine variables: {num=2}
```

Streams



```
numbers.stream( ).filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)
```

Streams

- It is a sequence of data that supports sequential or parallel aggregation operations, like:
  - Calculating the sum of its elements
  - Mapping the name of all people included in a list to their age

- Similar to a collection, but:
  - Designed for **declarative**/functional programming (in contrast to the more imperative code of collections).
  - Support **internal iteration**.
  - **They do not have storage**: they take the data from a source on demand.
  - They can represent an **infinite sequence**.
  - Designed to facilitate the **parallelization** of operations.
  - They support lazy **operations**.
  - Ordered Streams (e.g, ordered data source, like a List, or because they have been ordered with sort) and unordered.

Internal vs external iteration

- External iteration:

```
List<Integer> numbers =
        Arrays.asList(1, 2, 3, 4, 5);

int sum = 0;
for (int n : numbers) {
  if (n % 2 == 1) {
    int square = n * n;
    sum = sum + square;
  }
}

System.out.println(sum);
```

- The client extracts the elements, iterates them, and applies an algorithm to them.

- Internal iteration:

```
List<Integer> numbers =
     Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
           .filter(n -> n % 2 == 1)
           .map(n -> n * n)
           .reduce(0, Integer::sum);

System.out.println(sum);
```

- The client passes the algorithm to the stream.
- The stream applies the algorithm, iterating internally.

Internal vs external iteration

Parallelization

- External iteration:

```
List<Integer> numbers =
        Arrays.asList(1, 2, 3, 4, 5);

int sum = 0;
for (int n : numbers) {
  if (n % 2 == 1) {
    int square = n * n;
    sum = sum + square;
  }
}

System.out.println(sum);
```

- The client extracts the elements (for), iterates them, and applies an algorithm.

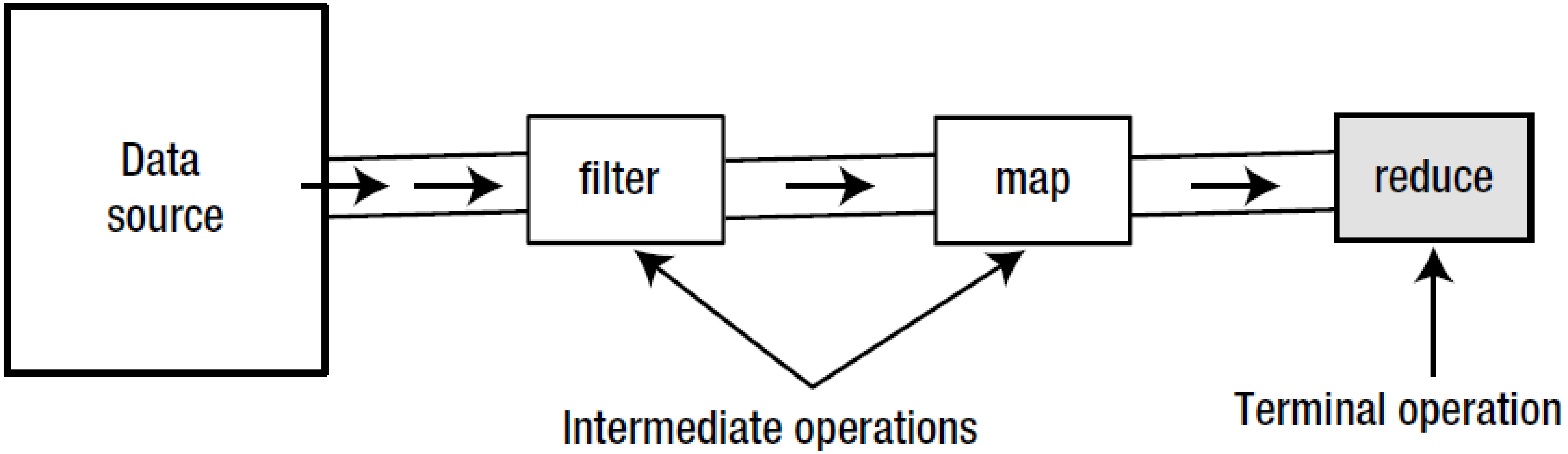
- Internal iteration:

```
List<Integer> numbers =
      Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.parallelStream()
            .filter(n -> n % 2 == 1)
            .map(n -> n * n)
            .reduce(0, Integer::sum);

System.out.println(sum);
```

- We can parallelize the operations on the stream to take advantage of multicore architectures.

Operating over streams

- Two kinds of operation
  - Interaveragete operations, which are lazy.
  - Terminal operations, which are eager.
- A lazy operation does not extract the elements of the stream until an eager operation is called
- Chain of lazy operations applied to a stream
  - Each one of them produces a new stream
- The terminal operation extracts the inputs of the stream, starts the computation and produces a result



Creating streams

- Can be created:
  - Empty
  - From values
  - From functions
  - From arrays
  - From collections
  - From files
  - From other sources

From values

- <T> Stream<T> of(T t)
- <T> Stream<T> of(T...values)

```
// Calculate the average number of letters in the word of a String
Stream<String> words = Stream.of("Java 8 is super".split("\\s+"));

OptionalDouble average = words.mapToInt(String::length).average();

System.out.println("average: "+average.orElse(0.0));
```

From values

- <T> Stream<T> of(T t)
- <T> Stream<T> of(T...values)

```
// More statistics: it is not possible to re-process a stream
long    words = Stream.of("Java 8 is super".split("\\s+")).
                          collect(Collectors.counting());
long    sum   = Stream.of("Java 8 is super".split("\\s+")).
                          mapToInt(String::length).sum();
OptionalInt    min   = Stream.of("Java 8 is super".split("\\s+")).
                          mapToInt(String::length).min();
OptionalInt    max   = Stream.of("Java 8 is super".split("\\s+")).
                          mapToInt(String::length).max();
OptionalDouble avg   = Stream.of("Java 8 is super".split("\\s+")).
                          mapToInt(String::length).average();

System.out.println(words+" "+sum+" "+
                   min.orElse(0)+" "+max.orElse(0)+" "+avg.orElse(0.0));
```

From values

- <T> Stream<T> of(T t)
- <T> Stream<T> of(T...values)

```
IntSummaryStatistics statistics =
      Stream.of("Java 8 is super".split("\\s+")).
             collect(Collectors.summarizingInt(String::length));

System.out.println(statistics);

// Output:
// IntSummaryStatistics{count=4, sum=16, min=1, average=4,000000, max=9}
```

From functions

- <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
- <T> Stream<T> generate(Supplier<T> s)

```
Stream.iterate(0L, n -> n + 2)    // infinite stream [0, 2, 4,…)
  .limit(5)                       // get the first 5
  .forEach(System.out::println); // print them

Stream.generate(Math::random)    // infinite stream of random numbers
  .limit(5)                      // get the first 5
  .forEach(System.out::println);

Stream.iterate(0L, n -> n + 2)
  .skip(100)                     // skip 100
  .limit(5)
  .forEach(System.out::println);
```

From arrays and collections

- `Arrays.stream`
- Methods `stream()` and `parallelStream()` over collections

```java
IntStream    numbers = Arrays.stream(new int[]{1, 2, 3});

Stream<String> words =
              Arrays.asList("this", "is", "converted", "into",
                             "list").stream();
```

Optional values (Optional)

- The value null se is used to represent the absence of a value
- Dificults the concatenation of operations over a stream
- `Optional<T>` represents a value of type T that can be null.
  - □ `isPresent()`: true if not null
  - □ `ifPresent(Consumer<? super T> action)`: executes the lambda if not null

```java
// Create an Optional for the string "Hello"
Optional<String> str = Optional.of("Hello");
// Print the value in the Optional, if present
str.ifPresent(value -> System.out.println("Optional contains " + value));
```

Some operations over Streams (1/2)

Operation	Type	Description
distinct	intermediate	Returns a stream with the different elements
filter	intermediate	Returns a stream with the elements satisfying the predicate
flatMap	intermediate	Returns a stream with the result of applying a function over the elements of the stream. The function produces a stream for each element, which is then flattened
limit	intermediate	Returns a stream of length less or equal to the limit passed as parameter
map	intermediate	Returns a stream with the result of applying the function to each element of the stream
peek	intermediate	Returns this stream, but applies an action when consuming elements (useful for debug)
skip	intermediate	Discards the n first elements and returns a stream with the following ones
sorted	intermediate	Returns a stream ordered by natural order, or by a Comparator

Algunas operaciones sobre Streams (2/2)

Operation	Type	Description
allMatch	terminal	Returns true if all elements of the stream satisfy the predicate
anyMatch	terminal	Returns true if some element of the stream satisfies the predicate
findAny	terminal	Returns an element of the stream. Returns an empty Optional if the stream is empty.
findFirst	terminal	Returns the first element of the stream (if the stream is unordered, it may return any element)
noneMatch	terminal	Returns true if no element of the stream satisfies the predicate
forEach	terminal	Applies an action to each element of the stream
reduce	terminal	Applies a reduction operation that calculates a unique value for the stream

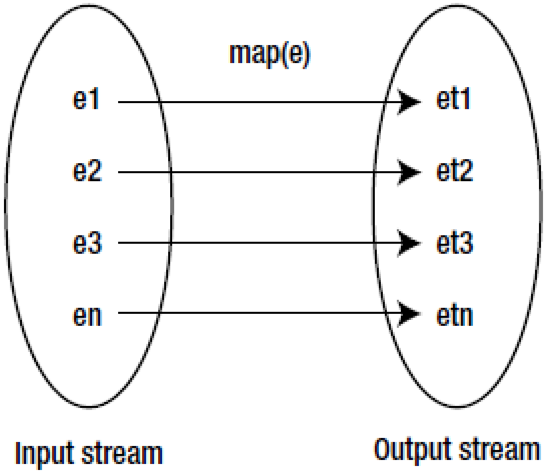
Example of debug

```
int sum = Stream.of(1, 2, 3, 4, 5)
          .peek(e -> System.out.println("Taking integer: " + e))
          .filter(n -> n % 2 == 1)
          .peek(e -> System.out.println("Filtered integer: " + e))
          .map(n -> n * n)
          .peek(e -> System.out.println("Mapped integer: " + e))
          .reduce(0, Integer::sum);

System.out.println("Sum = " + sum);
```

```
Taking integer: 1
Filtered integer: 1
Mapped integer: 1
Taking integer: 2
Taking integer: 3
Filtered integer: 3
Mapped integer: 9
Taking integer: 4
Taking integer: 5
Filtered integer: 5
Mapped integer: 25
Sum = 35
```

map operation

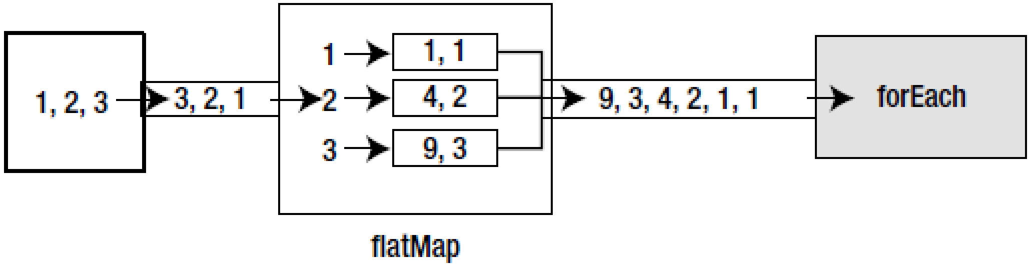
- Applies a function to each element of the stream
- Specialized versions, returning subclasses of Stream:
  - □ <R> Stream<R> map(Function<? super T,? extends R> mapper)
  - □ DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
  - □ IntStream mapToInt(ToIntFunction<? super T> mapper)
  - □ LongStream mapToLong(ToLongFunction<? super T> mapper)



```
IntStream.rangeClosed(1, 5)
   .map(n -> n * n)
   .forEach(System.out::println);
```

flatMap operation

- Applies a function that produces a stream to each element of the stream
- Flattens the resulting stream of streams



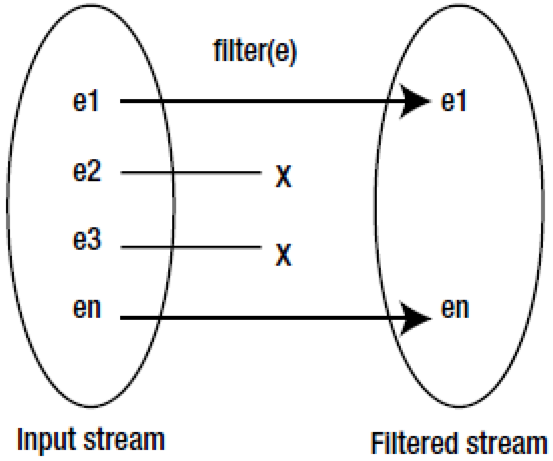
```
Stream.of(1, 2, 3)
    .flatMap(n -> Stream.of(n, n * n))
    .forEach(System.out::println);
```

Output:

```
1
1
2
4
3
9
```

filter operation

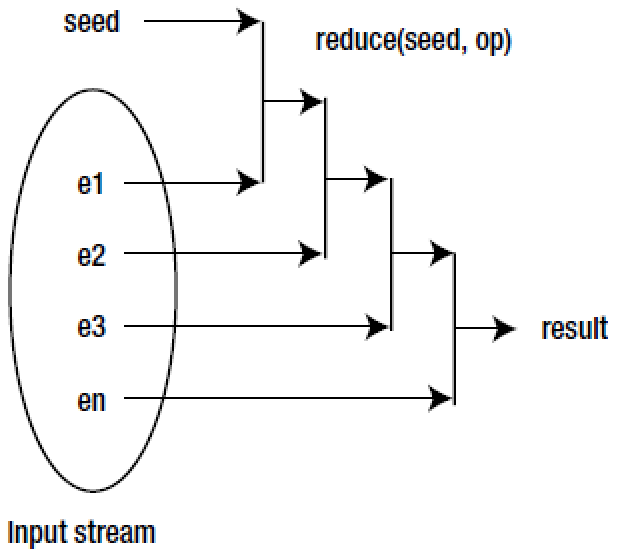
- Produces a stream with the elements satisfying the predicate



```
class Product{
  private int price;
  private String name;
  public Product(int p, String n) {
        this.price = p;
        this.name = n;
  }
  public int getPrice() { return this.price; }
  public String getName() { return this.name; }

  Stream<Product> sprod = Stream.of(  new Product(20, "Salt"),
                                      new Product(40, "Sugar"),
                                      new Product (5, "Wine"));
  sprod.filter( s -> s.getPrice() > 10).
      map(Product::getName).
      forEach(System.out::println);
  // Output: Salt Sugar
```

reduce operation

- Combines all elements of the stream into a unique value
- Takes an initial value and an accumulator
- The third parameter is a combinator lambda to combine the results of multiple threads in case of parallel execution



```
String conc = Stream.of("going ", "to ", " concatenate").
                        reduce("Concatenation: ", String::concat);

System.out.println(conc); // Output:  Concatenation: going to concatenate
```

Collectors

- Used when we need to save into a Collection the results of operating a Stream
- Or we need to apply complex logic when summarizing the information in a stream
  - □ <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
  - □ <R,A> R collect(Collector<? super T,A,R> collector)

```
List<String> names =
    sprod.map(Product::getName).
          collect(ArrayList::new, ArrayList::add, ArrayList::addAll);

System.out.println(names);
// Output:  [Salt, Sugar, Wine]
```

Collectors

- Used when we need to save into a Collection the results of operating a Stream
- Or we need to apply complex logic when summarizing the information in a stream
  - <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
  - <R,A> R collect(Collector<? super T,A,R> collector)

```
List<String> names =
    sprod.map(Product::getName).
          collect(Collectors.toList());  // equivalente a lo anterior

System.out.println(names);
// Output:  [Sal, Sugar, Wine]
```

Grouping data into Maps

- groupingBy(Function<? super T,? extends K> classifier)
- groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

```
                 class Product{
                   enum ProductType {FOOD, DRINK}
                   private ProductType tp;
                   private int price;
                   private String name;
                 …}
Stream<Product> sprod = Stream.of(
                  new Product(20, "Salt", ProductType.FOOD),
                  new Product(40, "Sugar", ProductType.FOOD),
                   new Product (5, "Wine", ProductType.DRINK));

Map<ProductType, List<Product>> prodsByType =
    sprod.collect(Collectors.groupingBy(Product::getType));

System.out.println(prodsByType);
// Output: {DRINK=[Wine (5€)], FOOD=[Salt (20€), Sugar (40€)]}
```

Grouping data into Maps

- groupingBy(Function<? super T,? extends K> classifier)
- groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

```
                    class Product{
                      enum ProductType {FOOD, DRINK}
                      private ProductType tp;
                      private int price;
                      private String name;
                    …}
Stream<Product> sprod = Stream.of(
                  new Product(20, "Salt", ProductType.FOOD),
                  new Product(40, "Sugar", ProductType.FOOD),
                   new Product (5, "Wine", ProductType.DRINK));

Map<ProductType, Long> prodsByType =
    sprod.collect(Collectors.groupingBy(Product::getType, Collectors.counting()));

System.out.println(prodsByType);
// Output: {FOOD=2, DRINK=1}
```

Grouping data into Maps

- groupingBy(Function<? super T,? extends K> classifier)
- groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)

```
Stream<Product> sprod = Stream.of(
                 new Product(20, "Salt", ProductType.FOOD),
                 new Product(40, "Sugar", ProductType.FOOD),
                  new Product (5, "Wine", ProductType.DRINK));

Map<ProductType, List<String>> prodsByType =
      sprod.collect(Collectors.groupingBy(Product::getType,
                   Collectors.mapping(Product::getName,
                                     Collectors.toList())));
System.out.println(prodsByType);
// Output: {DRINK=[Wine], FOOD=[Salt, Sugar]}
```

Index

- New concepts in interfaces
- Lambda expressions
- **Exercises**
- Conclusions and bibliography

Exercises (1/6)

- Given the following program to handle orders:

```
package products;

public enum ProductType {
   FOOD (6), TOBACCO (21), ALCOHOL (21);

   private double vat;

   private ProductType(double imp) { this.vat = imp;}

   public double getVat() { return this.vat;}
}
```

Exercises (2/6)

```
package products;

public class Product {
  private String name;
  private double price;
  private ProductType type;

  public Product(String n, double p, ProductType t) {
    this.name = n; this.price = p; this.type = t;
  }

  public double price() { return this.price*(1+this.type.getVat()*0.01);}

  @Override public String toString() {
    return this.name+" ("+this.price+", "+this.type.getVat()+"%)";
  }

  public ProductType getType() { return this.type; }
}
```

Exercises (3/6)

```
package products;
import …;

public class Order {
  private Map<Product, Integer> order =
    new LinkedHashMap<Product, Integer>();

  public Order addItemOrder(int n, Product p) {
    this.order.put(p, n); return this;
  }
  @Override public String toString() {
    return this.order.toString();
  }
}
```

Exercises (4/6)

```java
public static void main(String[] args) {
  Product p1 = new Product("Olives", 2, ProductType.FOOD);
  Product p2 = new Product("Beer", 1, ProductType.ALCOHOL);
  Product p3 = new Product("Ducados", 4, ProductType.TOBACCO);
  Product p4 = new Product("Chips", 1.5, ProductType.FOOD);
  Product p5 = new Product("Ham", 10.5, ProductType.FOOD);

  Order order = new Order().
                    addItemOrder(1, p1).
                    addItemOrder(5, p2).
                    addItemOrder(2, p4).
                    addItemOrder(1, p5);

  System.out.println(order);
}
```

```
{Olives (2.0, 6.0%)=1, Beer (1.0, 21.0%)=5, Chips (1.5, 6.0%)=2, Ham (10.5, 6.0%)=1}
```

Exercises (5/6)

- Modify class `Order`, to allow:
  - Obtain the products grouped by type (food, alcohol, tobacco)
  - Obtain the number of products of each type
  - Obtain the price itemized by type
  - Obtain the total order (using stream)
  - Obtain the total price of the products satisfying a condition

Exercises (6/6)

```
public static void main(String[] args) {
   …
   System.out.println("Order by type: "+
                            order.ProductsByType());
   System.out.println("Number of elements by type: "+
                            order.totalProductsByType());
   System.out.println("Itemized price by type: "+
                            order.totalPriceByType());
   System.out.println("Cost of products with net price bigger than 1 euro: "+
                            order.total( p -> p.price() > 2));
}
```

```
{Olives (2.0, 6.0%)=1, Beer (1.0, 21.0%)=5, Chips (1.5, 6.0%)=2, Ham (10.5, 6.0%)=1}
 Total: 22.48 €
Order by type : {ALCOHOL=[Beer (1.0, 21.0%)], FOOD=[Olives (2.0, 6.0%), Chips (1.5, 6.0%), Ham
(10.5, 6.0%)]}
Number of elements by type: {ALCOHOL=1, FOOD=3}
Itemized price by type: {ALCOHOL=6.05, FOOD=16.43}
Cost of products with net price bigger than 1 euro: 13.25
```

(Class Order)

Solution

```
public Map<ProductType, List<Product>> ProductsByType() {
   return this.order.keySet().stream().
              collect(Collectors.groupingBy(Product::getType));
}

public Map<ProductType, Long> totalProductsByType() {
   return this.order.keySet().stream().
               collect(Collectors.groupingBy(Product::getType, Collectors.counting()));
}

public Map<ProductType, Double> totalPrecioByType() {
   return this.order.keySet().stream().
              collect(Collectors.groupingBy(Product::getType,
                         Collectors.summingDouble( (Product p) ->
                                                   p.price()*this.order.get(p))));
}
```

Solution

(Class Order)

```
public double total() {
   return this.order.keySet().stream().
                      mapToDouble( p -> p.price()*this.order.get(p) ).
                      sum();
}

public double total(Predicate<Product> pred) {
   return this.order.keySet().stream().
                     filter(pred).
                     mapToDouble( p -> p.price()*this.order.get(p) ).
                     sum();
}
```

Index

- New concepts in interfaces
- Lambda expressions
- Exercises
- **Conclusions and bibliography**

Conclusions

- Lambda expressions introduce flexibility and conciseness in specifying operations with collections of elements
- Other advantages, such as ease of parallelization

- We have **NOT** covered other advanced aspects:
  - The API of functional interfaces and streams is extensive
    - Part of this API will be explored and practiced in the lab assignment.
    - The functional paradigm (lisp) will be studied in more detail in the Artificial Intelligence course
  - Design of embedded domain specific languages:
    - In Java using lambdas: (http://en.wikipedia.org/wiki/Domain-specific_language)
    - The design of languages embedded in Ruby will be studied in the optional course automated software development.

Bibliography

- Java 8 Lambdas. Functional Programming for the masses. O’Reilly. Richard Warburton. 2014.

- Beginning Java 8 Language Features. Kishori Sharan. Apress. Agosto 2014.

- Functional Programming in Java. Harnessing the power of Java 8 Lambda Expressions. The Pragmatic Programmers. V. Subrmanian. 2014.