UNIVERSITÀ DEGLI STUDI DI TORINO

Polo delle Scienze della Natura

Corso di Laurea Triennale in Informatica

Tesi di Laurea Triennale

Introducing Java 8

Relatore:

Prof. Viviana Bono

Candidato:

Enrico Mensa

Anno Accademico 2012/2013

Ringraziamenti

La vita è come una somma.

Grazie ai miei genitori, alle persone che ho più vicino,
al mio impegno, ai miei docenti, alla musica,
per essere stati tutti ottimi addendi.

Spero, con quegli addendi, di aver fatto una buona somma.

Contents

Chapter 1

Introduction

Java is a programming language that was born in 1995 from the mind of James Gosling. At first it was supposed to be used for the development of little elettrical engines, then, with the spread of the Internet, Java started to seem a good solution for programming the web. Since that moment a lot of changes have been done to the language: now we are waiting for the 8th version, expected in March 2014.

This document wants to show some of the new features that will be introduced in *Java 8*, reconnecting them to the already present functionality, with a special attention to the topic of *code reusability.* As Java 8 brings some deep class–related changes, we are going to see not only what the new features offer but also where the big project of *Java* is moving to.

1.1 What's in this document

Let's see briefly all the topics that we are going to treat in this document. The document is divided in two big parts. The first part will introduce most of the features of Java 8: Lambda Expressions & Virtual Extensions Methods. At the beginning we will see the motivations that guided Oracle's programmers in the develop of those: this will be very important to understand some of the implementation details that we are going to see.

After that, we will discuss the various kind of classes that Java offers, trying to understand why using a certain implementation is better than using another one. We will discuss a lot about *anonymous classes*, and how their use is totally renewed in Java 8. In the second part of the document we will try to use the new features to propose a trait-oriented programming style, with the goal of *code reusability*: but we don't want to anticipate too much!

1.2 Goals and motivations of JEP 126

JEP 126 (acronym for JDK Enhancement Proposal) is the assigned name of *Lambda Expressions & Virtual Extensions Methods* proposal. JEP 126 is a follower process of the Project Lambda (that has the corrisponding JSR 335).

In a nutshell, the intent is to add lambda expressions (more exactly *closures*, we will specify it later) to the Java language: by doing this it is obviusly necessary to add some related features such as method references, enhanced type inference and virtual extension methods (necessary for preserve backward compatibility).
It seems like we have a lot of stuff to talk about!

1.2.1 Why Lambda Expressions?

Lambda expressions are fundamentally anonymous functions.
First of all, we have to clarify the two different concepts of *closures* and *lambda expressions.* Closures are functions that have access to all the variables in their environment (lexical scope) as well as the parameters of the function itself. On the other side, lambda expressions do not have access to

the lexical scope but only to the parameters. If this definition sounds fuzzy do not worry: thanks to further examples will be very easy to see that Java implements lamba expressions as, technically, closures. However, we will use both the terms as synonyms for simplicity.

Lambda expressions are already included in a lot of object-oriented programming languages such as Scala and Ruby, so a lot of programmers are becoming familiar with this powerful feature and with the programming models it unlocks.

One of the most relevant uses of lambda expressions is their application in the *internal iteration* idiom. Actually, if you want to iterate an array or a collection you have to write your own loop in the code: this means that the logic control of the iteration lives **outside the structure**. A classical example could be a foreach looping an array.
This kind of approach brings a direct conseguence: it is impossibile to have a multi-core iteration! But thanks to lambda expressions we could write the iteration code **inside the structure** and then pass a piece of code (that is our lambda expression) to be executed while looping. Trasferring the loop logic to the data structure (that of course 'knows' much better itself than the external user) brings the opportunity to choose a better scheduling of the computation (for example by alternating the execution order) so that we can have a raise of the performances. Lambda expressions are also important because they help to write slim code: we will see that a lambda expression has a very pithy sintax but it is also very expressive.

1.2.2 Why Virtual Extension Methods?

As we have just said, lambda expressions are great. But to reach all their potential, lambda expression cannot be implemented without the help of virtual extension methods.

Let's use an example to explain the problem. We have the interface Collection<T> that is actually present in Java 7. We can think that a lot of other classes implement the Collection<T> (even in the SDK itself).

So now, with lambda expressions, we would expect to have a forEach(LE) method that takes a lambda expression as argument. But adding this method to Collection<T> interface would mean break backward compatibility: in other words, every class implementing Collection<T> interface would have to change by adding a forEach(LE) method.

Oracle's developers solved this problem by introducing *Virtual Extension Methods* (aka Default Methods, aka Defender Methods). A default method is a virtual method (just as any other method in Java) that specifies a concrete implementation: if any class implementing the interface will override the method, the more specific implementation will be executed. But if the default method isn't overrided, then the default implementation written in the interface will be executed.

This is a huge change for the whole class system and we are going to discuss the pros and cons of this choice.

Chapter 2

Exploring Java 8

2.1 Nested Classes

Let's start by refreshing some notions about a feature that is already provided in Java since long times: *nested classes*. A nested class is, banally, a **class that is defined within another class**.

```java
public class OuterClass {
  // ...
  public class NestedClass {
    // ...
  }
}
```

Listing 2.1: An example of a nested class.

Terminology: a nested class is called *inner class* if it is a non-static class (otherwise is just a nested class).

If we have a static nested class we can refer to it by the use of the notation OuterClass.NestedClass. Of course we don't have the this variabile and we cannot refer to the OuterClass fields from the inside of NestedClass. It is just like having an external class, but it is nested for packaging reasons.

If we have an inner class, that class will be binded to the outer class. For example an instance of an InnerClass object can be created in this way:

```
OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```

Listing 2.2: An example of creation of an InnerClass object.

Inner classes cannot define any static member: for this reason also inner interfaces do not exist at all (because they are inherently static).

There are several reasons for using a nested class:

- Incapsulation is favorited.
- Code is more readable.
- The logic of the code is more clear.

2.1.1 Inner Classes: Local & Anonymous Classes

Local Classes

Sometimes we need a class for a specific purpose and we know that such class would be used just in that certain context. So it is possibile to create a class inside any block (a method body, for example) of code.

```
public class LocalClassExample {

    static String regularExpression = "[^0-9]";

    public static void validatePhoneNumber(
      String phoneNumber1, String phoneNumber2) {

        final int numberLength = 10;

        class PhoneNumber {

            String formattedPhoneNumber = null;

            PhoneNumber(String phoneNumber) {

                String currentNumber = phoneNumber.replaceAll(
17                    regularExpression, "");
18                if (currentNumber.length() == numberLength)
19                    formattedPhoneNumber = currentNumber;
20                else
21                    formattedPhoneNumber = null;
22            }
23
24            public String getNumber() {
25                return formattedPhoneNumber;
26            }
27
28        }
29
30        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
31        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);
32
33
34        if (myNumber1.getNumber() == null)
35            System.out.println("First␣number␣is␣invalid");
36        else
37            System.out.println("First␣number␣is␣" +
38              myNumber1.getNumber());
39        if (myNumber2.getNumber() == null)
40            System.out.println("Second␣number␣is␣invalid");
41        else
42            System.out.println("Second␣number␣is␣" +
43              myNumber2.getNumber());
44
45    }
46
47    public static void main(String... args) {
48        validatePhoneNumber("603-555-0123", "555-0123");
49    }
50 }
```

Listing 2.3: An example of a local class from The Java Tutorials.

As we can see, the class LocalClassExample has a static method validatePhoneNumber that takes two strings as parameters. Every string is processed by the constructor of the local class PhoneNumber invoked in lines 30 and 31.

A local class have access to:

- Fields of its enclosing class (line 17).

- All *final* variables of the block where it is located (in Java 8 also *effectively final* variables are used: an effectively final variabile is a variable that isn't declared as final but doesn't change his value after the first assignment).
- From Java 8, also parameters of its enclosing method are accessible.

It is interesting to discuss a little bit more about the second point. Let's say that at line 8 we remove the `final` keyword: in Java 8 this code will compile and execute correctly as `numberLength` variable is effectively final. But then, if we add into the `PhoneNumber` constructor an assignment for the `numberLength` variable, we will get an error during compilation: "local variables referenced from an inner class must be final or effectively final".

About static elements. Let's make clear some points about static methods/fields/classes. Local classes that are defined **in a static method** (like is `PhoneNumber` in our example) can only refer to static members of the enclosing class (and that's why `regularExpression` is declared `static`).

A local class is an inner class so it can't be a static class (thus it is impossibile to create "local interfaces", as interfaces are inherently static). As a consequence of this fact, a local class can't use most of static declarations. You cannot declare static methods in a local class. Let's see this method:

```java
public void printWelcome {
    class Printer {
        public static void printWelcome() {
            System.out.println("Welcome to this document.");
        }
    }
    Printer.printWelcome();
}
```

Listing 2.4: A static method in a local class: this code won't compile.

This code doesn't compile because printWelcome() method is declared static. However a local class can have static members if they are constant variables. A *constant variable is a variable of primitive type or type String that is declared final and initialized with a compile-time constant expression* [Ora] (a constant expression such a string or an arithmetic expressions that is evaluable at compile-time). To make the previous code compile we have to do some changes.

```
public void printWelcome {
    class Printer {
      public static final String message =
        "Welcome to this document.";
        public void printWelcome() {
            System.out.println(message);
        }
    }
    Printer p = new Printer();
    p.printWelcome();
}
```

Listing 2.5: Thanks to the constant variable this code will compile.

Anonymous Classes

Anonymous classes are very helpful as they make code more slim and concise: an anonymous class is declared and instatiated at same time.
Just as the name itself says, anonymous classes do not have a name so they are "one-shot" classes, perfect for a single use.
The similarity between anonymous classes and local classes is very strong, except the fact that local classes are declared by the use of a *class declaration* while anonymous classes are declared by the use of an *expression.*

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

    public class Subscribers extends JFrame {
```

```
 7
 8      private DefaultListModel subscribers;
 9
10      public Subscribers() {
11
12         //...
13
14         JButton addButton = new JButton("Subscribe␣user");
15         addButton.addActionListener(
16           new ActionListener() {
17              public void actionPerformed(ActionEvent event){
18                      //Particular implementation
19              }
20           });
21
22         //...
23
24         setDefaultCloseOperation( EXIT_ON_CLOSE );
25         pack();
26         setVisible( true );
27      } // end constructor
28
29      public static void main( String args[] ){
30         new Subscribers();
31      }
32
33 }
```

Listing 2.6: An example of an anonymous class.

At line 15 we see the use of an anonymous class: ActionListener is an interface that has just one method called actionPerformed(). Therefore we are declarating a new class that implements ActionListener interface but we are not giving a name to it. We are just getting a pointer to an object that has the implementation written between the brackets (from line 16 to line 20). As we can see, at line 20 we have a semi colon because the declaration/instantiation is an expression (like we already said).

To declare/instatiate an anonymous class we use an expression that consist of four elements:

- The new operator,

- The name of an interface to implement or a class to extend (in this case, we had the ActionListener interface),

- Round parentheses wrapping the arguments for the constructor (in case of interface we cannot have arguments, as interface are constructor-less, so parentheses will be empty),

- A body surrounded by brackets.

Access to local variables. Anonymous classes and local classes have the same kind of access to local variables. An anonymous class have access to the members of its enclosing class, to all variables of the enclosing class as long as they are final (or effectively final in Java 8) and can provide static member only if those are constant variables. Inside an anonymous class you can declare fields, extra methods, instance inizializers and local classes but it is not possibile to declare any constructor (it is obvious as you wouldn't even be able to name the class and so calling it).

2.2 New anonymous classes: Lambda Expressions

It is finally time to talk about something new that Java 8 brought to us: lambda expressions. As we already mentioned a lambda expression is a special kind of anonymous class. Let's take a look back to Listing 2.6: as the goal of an anonymous class is to write slim and concise code, it looks like we are wasting a lot of lines to declare just a function.
So Oracle's developers decided to introduce a more concise way to write such kind of code (a single-function class): lambda expressions.

2.2.1 Understanding lambdas: an use case

To make clear why lambda expressions can be so userful, we are going to deep in one real-life use case (taken from the Java Tutorials [Ora]).

We are building a social network, here there's an use case:

Admin	**System**
Selects a certain research criteria for users.	-
Selects a certain action to be applied at selected users.	-
Submits the request.	-
-	Finds all users fitting the research criteria.
-	Applies the selected action to all those users.

Inside the social network, all members are collected in a List<Person>. Here's the Person object:

```java
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

Listing 2.7: How it is represented a member inside the social network.

We are going to see seven different approach to get a solution for this problem. At first we will only think to the search-approach while trying to execute a print action. Only at the end we will see that also actions can be treated in a more intelligent way.

Approach 1: One method for each charateristic

A very easy approach is to create one method for each search criteria.

```java
public static void printPersonsOlderThan(
    List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Listing 2.8: This method prints users selected by an age search-criteria.

As a direct conseguence of this choice, we are going to have a lot of different tricky-named methods. Also, changing the structure of the Person class would be risky (perhaps the age could be stored in a different way?). This kind of approach breaks the low coupling[1] principle and we do not want this.

Approach 2: More general search methods

A good way to make things more generic is to create methods that can work in multiple cases. For example, we could change the previous method by selecting users within an age range.

```java
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
```

[1]Low coupling is one of the GRASP patterns. It promotes lower dependency between classes to have an higher reuse potential. For a more detailed reading we suggest [Lar04].

```
    for (Person p : roster) {
        if (low >= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Listing 2.9: A more general method for selecting users by age.

We didn't really solve the previous problem. What if we want to select users by sex? Or by email address? What happens if we want to combinate more research criterias? We will still have a lot of methods.

Approach 3: Define separate search criteria in a local class

Let's see a more interestring approach. It could be a nice idea to use a local class defining the test logic somewhere else and so keeping distant the action code and the test code.

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Listing 2.10: This code doesn't specify any search criteria.

Then we define an interface that every search criteria will have to implement.

```
interface CheckPerson {
    boolean test(Person p);
}
```

Listing 2.11: The generic search criteria interface.

We have nothing else to do than create a single local class for every search criteria.

```
class CheckPersonEligibleForSelectiveService
            implements CheckPerson {
```

```java
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Listing 2.12: This class selects users that are eligible for Selective Service.

Now it will be very easy to call the printPersons method for different search criterias thanks to local classes.

```java
printPersons(roster,
        new CheckPersonEligibleForSelectiveService());
```

Listing 2.13: How to use the printPersons method.

The logic behind our code is now so much clearer (also the problem of the mutability of Person is solved) but we have a lot of additional code: every search criteria needs a local class and we have also to define a certain number of interfaces.

Approach 4: Define separate search criteria in an anonymous class

In the last approach we used a local class that implements an interface. But as we have seen in Section 2.1.1 we have an other option to do so. We could use instead of a local class an anonymous one and pass it as a parameter to the printPersons method.

```java
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Listing 2.14: We define the anonymous class while calling the method.

We are close to a good solution: the code is slim, we do not have anymore to declare one class for each criteria search, we just define the criteria directly while calling the method.
But we can do something better: doesn't all the anonymous class definining expression look to much redundant?

Approach 5: Define separate search criteria in a lambda expression

As we are fundamentally defining a little method, why do not use a lambda expression?

Let's introduce some terminology: the CheckPerson interface is defined as a *functional interface* as it only contains one method.
More precisely a functional interface is an interface that has exactly one explicitly declared **abstract** method[2][3]. It is important remember the precise definition as a functional interface can still have many default methods (we are going to talk a lot about those in further chapters). Anyway, given the fact that a functional interface has only one abstract method, we can omit his name and so have a very much slimmer code.

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Listing 2.15: A lambda expression is given to printPersons method.

We are going to talk about the syntax of the lambda expressions later (in Section 2.2.2).

[2]An abstract method is a method that lacks a body. Abstract modifier doesn't have to be explicitally written in the signature of the method, as if it doesn't have a body it is by definition an abstract method.
[3]Those interfaces are also called SAM Interfaces (acronym for Single Abstract Method interfaces).

Approach 6: Use standard functional interfaces with lambdas

Lambda expressions are given in Java 8 together with a bunch of standard functional interfaces. Those are very helpful in our goal to avoid extra code: for example the CheckPerson interface is just the same of the Predicate<T> interface. Reconsider Listing 2.11: CheckPerson is basically a functional interface with one method that, given one parameter (of a certain type), returns a boolean value.

```java
interface Predicate<T> {
    boolean test(T t);
}
```

Listing 2.16: The Predicate<T> given in java.util.function since Java 8.

Thanks to the generics[4] system we can use as many lambda expressions as we want without declaring a single custom functional interface. By the use of the Predicate<T> interface we can modify our print method.

```java
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for(Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Listing 2.17: We are using the Predicate<Person> instead of our CheckPerson.

Meanwhile nothing has changed about the way to call it:

```java
printPersonsWithPredicate(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Listing 2.18: We have the same lambda expression we used before.

[4]See http://docs.oracle.com/javase/tutorial/java/generics/ for more details.

That's perfect! We have totally removed all the redundant lines and we have an easy-to-change system for selecting the users of the social network. Low coupling and high choesion patterns are followed at best.

Approach 7: An intelligent use of lambda expressions

In the whole process of resolving the problem we only considered one action: printing. But what if we want to perform another kind of action on the selected users? The problem is just the same as before, so, let's skip directly to the smartest solution: lambda expressions. Instead of writing directly the action to be performed, we could give another parameter to the method, a new lambda expression. Again it is a good idea to use a built-in functional interface. As we want to accept a new user, we will use the Consumer<T> functional interface that has one void method accept and takes one parameter of type T.

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }
}
```

Listing 2.19: A second lambda expression (Consumer<Person>) is used.

About the method invocation, we just need to add the second lambda expression to make the whole system work.

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

Listing 2.20: A second lambda expression is passed has parameter.

We finally did it. Now our search/action system is parametric in the two dimenions of search criterias and performing actions. Our code is clear and slim, it is easy to be expanded and there is no violation of the low coupling and high choesion patterns.

2.2.2 Syntax of a lambda expression

A lambda expression is composed by:

- a list of formal parameters enclosed in parentheses. Every parameter is separated from the other by a comma. It is possibile to omit the type of the parameters as well as the parentheses if the parameter is just one.

```
    p -> p.getGender() == Person.Sex.MALE

    (a, b) -> a.field1 > b.field2
```

Listing 2.21: Two examples of valid lambda expressions.

- the arrow token, ->.
- a body that can consist of a single expression or a statement block. In case of a single expression, Java evalutates it and returns its value. It is anyway possibile to specify the `return` keyword.

```
    p -> { return p.getGender() == Person.Sex.MALE }
```

Listing 2.22: A single expression body with explicit `return`.

In this example we enclosed the return block in brackets: this is necessary as a return statement isn't an expression, it is a block. However it's possibile to use void-methods without adding brackets just like in this example:

```
      p -> System.out.println(p.name)
```

Listing 2.23: A void method invocation doesn't need any bracket.

Example of lambda expressions application

Let's take a look of this example from Java Tutorials [Ora].

```
 1 public class Calculator {
 2 
 3     interface IntegerMath {
 4         int operation(int a, int b);
 5     }
 6 
 7     public int operateBinary(int a, int b, IntegerMath op) {
 8         return op.operation(a, b);
 9     }
10 
11     public static void main(String... args) {
12 
13         Calculator myApp = new Calculator();
14         IntegerMath addition = (a, b) -> a + b;
15         IntegerMath subtraction = (a, b) -> a - b;
16         System.out.println("40␣+␣2␣=␣" +
17             myApp.operateBinary(40, 2, addition));
18         System.out.println("20␣-␣10␣=␣" +
19             myApp.operateBinary(20, 10, subtraction));
20     }
21 }
```

Listing 2.24: A class that uses some lambda expressions.

This example shows us how lambda expressions can make our code easily expandable and easy-to-read. In fact we can see that both lines 14 and 15 define two different lambda expressions, used in line 16 and 17 as parameters for the operateBinary() method. The two lambda expressions are instances of the IntegerMath functional interface (lines 3-5).

2.2.3 Type of a lambda expression

Now that lambdas are well defined, one question should come to us: it is a lambda expression an object? And if it is, what type does it have?
Yes, lambda expressions are objects. Every lambda expression is an instance of a functional interface which is itself a subclass of `Object`. So, what's the type of a lambda expression? In most of the previous examples we didn't really specify any type, we just placed the lambda expression in his spot expecting that the compilator would undestand the type of the expression. That's exactly how it works! The type of a lambda expression is **deduced from the context** in which it is used.
As a conseguence lambda expressions can only be written in situations where the compilator can determine a target type:

- Variable declarations
- Assignments
- Return statements
- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, ?:
- Cast expressions

About parameters type, Java uses two other well known-features: overload resolution and type argument inference.
Let's consider two functional interfaces [Ora].

```

public interface Runnable {
    void run();
}

public interface Callable<V> {
    V call();
}
```

Listing 2.25: Method run() is void while method call() returns a V value.

Now, let's suppose that we have a method invoke that is overloaded.

```
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Listing 2.26: The second invoke() overloads the first one.

Which method will be executed if we write this line?

```
String s = invoke(() -> "done");
```

Listing 2.27: Example of call of invoke() method.

We are saving the return from the method call: this means that the invoke that will be executed is the second one as it has a return (of T type). So the T invoke(Callable<T> c) will be executed: this means the our lambda expressions has Callable<T> type.

2.2.4 Access to local variables

We still remember that lambda expressions are, basically, anonymous classes. The access to local variables is in fact the same as the anonymous classes one. We already spoke about that, so let's just take a look at this example from Java Tutorials [Ora].

```
import java.util.functions.Block;

public class LambdaScopeTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {

            // x = 99;

            Block<Integer> myBlock = (y) ->
            {
                System.out.println("x␣=␣" + x); // Statement A
                System.out.println("y␣=␣" + y);
                System.out.println("this.x␣=␣" + this.x);
                System.out.println("LambdaScopeTest.this.x␣=␣" +
                    LambdaScopeTest.this.x);
            };

            myBlock.apply(x);

        }
    }

    public static void main(String... args) {
        LambdaScopeTest st = new LambdaScopeTest();
        LambdaScopeTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

Listing 2.28: Example of access to local variables from a lambda expression.

We have this output:

```
x = 23
y = 23
this.x = 1
LambdaScopeTest.this.x = 0
```

Listing 2.29: Output generated from previous code.

Let's make two different considerations:

- If we remove the comments from line 13 (`x = 99`), we will have a compile time error in statement A (line 17). Just like in anonymous and local classes "*local variables referenced from a lambda expression must be final or effectively final*".
- If we give `x` as a parameter instead of `y` to the lambda expression (line 15) we will get an error from the compiler: as lambda expressions do not define any new scope, variable `x` can't be used because it is already defined.

2.2.5 Lambda expressions and recursion

Can we use a lambda expression for recursion purpoises? Yes, we can. Of course we have to declare the lambda expression separately: it needs a name to be recursional-called.

```
UnaryOperator<Integer> factorial =
     i -> {return i == 0 ? 1 : i * factorial.apply }
```

Listing 2.30: Example of recursion with a lambda expression.

2.2.6 Method and constructor references

Thanks to lambda expressions we can define a new function very comfortably. However, sometimes a lambda expression does nothing more than call an existing method. In this cases it is better to specify the name of the method that is called, to have a clearer reference to it. **Method references** help us in this purpose: they are compact lambda expressions for methods that already have a name. We will consider again an example from Java Tutorials [Ora].

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public Calendar getBirthday() {
        return birthday;
    }

    public static int compareByAge(Person a, Person b) {
        return a.birthday.compareTo(b.birthday);
    }}
```

Listing 2.31: The Person class we have already seen before.

Let's say that we want to sort an array of Person objects. We could create a new class implementing Comparator<T> and then give it as a parameter to Arrays.sort() method that has the following signature:

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

Listing 2.32: Signature of Arrays.sort() method.

But Comparator<T> is a functional interface so we could directly use a lambda expression directly:

```
Arrays.sort(rosterAsArray,
    (Person a, Person b) -> {
        return a.getBirthday().compareTo(b.getBirthday());
    }
);
```

Listing 2.33: The second parameter of Arrays.sort() is a lambda expression.

This works great... But we can do it better. We already have a method that does the comparation between two Person's birthdays, it is implemented in Person class; we are talking about compareByAge() method. So we can replace the previous code in this way:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person.compareByAge(a,b)
);
```

Listing 2.34: Instead of writing a new body we use an existing method.

To make it more concise, Oracle's programmers gave us *method references*, so we can write:

```
Arrays.sort(rosterAsArray,
    (a, b) -> Person::compareByAge);
);
```

Listing 2.35: Use of a method reference.

Kinds of method references

There are four different ways to use method references.

Type	**Example**
Reference to a static method	ContainingClass::staticMethodName
Reference to an instance method of a particular object	ContainingObj::instanceMethodName
Reference to an instance method of an arbitrary object of a particular type	ContainingType::methodName
Reference to a constructor	ClassName::new

Reference to a static method. The example (Person::compareByAge) we have just seen is a reference to a static method.

Reference to an instance method of a particular object. If we want to specify a non-static method from an instance instead that from a class, we can use this kind of method reference. So if we have a certain variable var with a method m, we can write var::m.

Reference to an instance method of an arbitrary object of a particular type. We can illustrate, for example:

```
Arrays.sort(rosterAsArray, String::compareToIgnoreCase);
```

Listing 2.36: Example of method reference.

Reference to a constructor. Imagine to have one method that transfers every element of a collection into another: let's call it transferElements(). The first parameter will be the collection with the elements to be transferred, while the second parameter is the new collection. We can do something like this:

```
Set<Person> roosterSet = transferElements(roster, HashSet::new);
```

Listing 2.37: Example of method reference.

We used the Supplier<T> functional interface for the second parameter: it has a method get that has no parameters and returns an object.

2.3 Every class has a right place

We introduced nested classes, local classes, anonymous classes and lambda expressions. So now, how do we use them? Why should we select a lambda expression instead of a local class? Let's define some general rules about the use of all these features.

- **Local class**: it is good to use when you will have multiple instances of the class, when you need access to the constructor or you need to introduce a new type.

- **Nested class**: You can use it in the same situations you would use a local class, but with this technique you can make the type more available (but of course you do not have any access to local variables/method parameters of the outer class).

- **Anonymous class**: Use it when you need an unique instance of a class that has more than one method, fields, etc.
- **Lambda expression**: Use it for encapsulating the logic of some kind of action that will be used in other code. It is good to use also when you need an instance of a functional interface (as we have seen in Section 2.2.1).

2.4 Standard functional interfaces

We already spoke about them in Section 2.2.1: a functional interface is an interface that has exactly one abstract method.
Let's take a brief tour in all the features given in Java 8 about standard functional interfaces. Some changes have been applied to already existing class and, as we know, other classes have been added to the SDK.

2.4.1 Functional interfaces from Java SE 7

Here's a list of some of the classes already in Java SE 7 that are very suitable to become functional interfaces in Java 8.

- java.lang.Runnable
- java.util.concurrent.Callable
- java.security.PrivilegedAction
- java.util.Comparator
- java.io.FileFilter

- java.nio.file.PatchMatcher
- java.lang.reflect.InvocationHandler
- java.beans.PropertyChangeListener
- java.awt.event.ActionListener
- javax.swing.event.ChangeListener

Thanks to Java 8 it will be so much easier to, for example, assign a listener to a GUI Object, define a thread and so on. We are using the exact same clasess as before but with a more concise and intelligent sintax.

2.4.2 New functional interfaces introduced in Java 8

As we discussed before a lot of actions that can be written inside a lambda expression could be re-grouped in a few more general functional interfaces. That's why Java 8 gives us some standard functional interfaces such as:

- Function<T, R>: takes a T as input and returns a R as output.
- Predicate<T>: takes a T as input and returns a boolean as output.
- Consumer<T>: takes a T as input and is void.
- Supplier<T>: with an empty input returns a T.
- BinaryOperator<T>: takes two T as input and returns a T.

Primitive specializations[5] for most of these interfaces are provided for int, long and double primitive types.
Just to make an example, we have IntConsumer interface, that works as a primitive alternative for the Consumer<T> functional interface.

[5] Having this kind of classes it is a common solution used in Java to wrap primitive types and avoid boxing/unboxing when inputs or outputs are primitive. Others well known examples are Integer, Double and Long classes.

2.5 Defaults Methods

Another big change to Java is given by the introduction of *default methods*[6]. Initially called 'defence methods' (their purpose was to preserve backward compatibily while introducing new features to the language), those methods can be added inside an interface. Their peculiarity is the fact that they provide an implementation for themselves.

This is a huge change: the concept behind an interface was to give a sort of "contract" to the user (which methods must be implemented), but not to specify any detail of the contract itself (any body of any method). Also, such a change has an impact on the compilator that now has to work on the hierarchy of the classes to find which methods are defaults and which are not, by selecting at the end the right set of methods that will be executed if called.

The main characteristic of default methods is that they are *virtual* like all methods in java, but they provide a default implementation in case that the implementing class doesn't provide a method body.

Having a client-view, extension methods are no different from ordinary interface methods: client does not have to care where the implementation comes from.

Let's see a first little example of default method just to taste the sintax.

```java
public interface A {
    default void foo(){
        System.out.println("Calling A.foo()");
    }
}
```

Listing 2.38: A basic example of a default method foo in interface A.

[6]The name used by Brian Goetz, responsable for this project, it is *Virtual Extension Methods*.

2.5.1 Method dispatch

The new feature makes necessary add a new phase in the *method dispatch algorithm*.

Before Java 8, if a method m is invoked on a A object we have this kind of approach: "Search in A if there is any non-abstract method m. If there is not, search in A's superclasses and so on until we reach Object." If Object is reached and no method m corresponding to the one that has been called have been found, we have a linkage exception.

But now, with default methods, we have this new algorithm: "Search in A and its superclasees if there is any (abstract or not) matching method m. In case we do not find any implementation for m, then we search for a *unique most specific interface* providing a default implementation for m among A's interfaces." If we do not find any interface that matches those characteristics or we find more than one, the linkage exception is thrown. We will report now the method resolution detail, as defined by Goetz [Goe11, GF12]:

1. First search for a matching method in the superclass hierarchy, from receiver class proceeding upward through superclasses to Object. If a non-abstrac method is found, the search is resolved successfully. If an abstract method is found, the search fails and extension methods are not considered in the resolution.

2. Construct the list of interfaces implemented by A, directly or indirectly, which contain a matching extension method.

3. Remove all interfaces from this list which is a superinterface of any other interface on the list (e.g., if A implements both Set and Collection, and both provide a default for this method, remove Collection from the list.)

4. If the resulting list of interfaces contains a single interface, the search is resolved successfully; the default implementation named in that interface is the resolution. If the resulting set has multiple items, then throw a linkage exception indicating conflicting extension methods.

Also, we can define four rules about method linkage [GF12]:

- A method defined in a type takes precedence over methods defined in its supertypes.
- A method declaration (concrete or abstract) inherited from a superclass takes precedence over a default inherited from an interface.
- More specific default-providing interfaces take precedence over less specific ones.
- If we are to link m() to a default method from an interface, there must a unique most specific default-providing interface to link to.

It is necessary keep in mind this rules because sometimes they can bring us strange behaviours: we will try to point them out.

2.5.2 Some examples of default methods

Let's see some examples to clarify what we have just learnt.

First example

Let's start with something simple. If the class that implements the interface using default methods doesn't override those methods, the default implementation provided in the interface will be executed.

```
interface A {
  default void m() {
    System.out.println("Hello from interface A");
```

```java
  }
}

class B implements A {} //doesn't override m

public class FirstDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

Listing 2.39: Simple example of use of default methods.

The output will be: "*Hello from interface A*".

Classes always win

We already knew it from the rules mentioned in Section 2.5.1: classes always win. If the implementing class overrides a default method from the implemented interface, the implementation provided in the class will be executed.

```java
interface A {
  default void m() {
    System.out.println("Hello from interface A");
  }
}

class B implements A {
  public void m() {
    System.out.println("Hello from class B");
  }

} //overrides m

public class SecondDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

Listing 2.40: Class B takes precedence over interface A.

The output will be: "*Hello from class B*".

Most specific interface wins

If no class overrides a default method, the default method with the most specific implementation will be executed.

```java
interface A {
  default void m() {
    System.out.println("Hello from interface A");
  }
}

interface B extends A {
  default void m() {
    System.out.println("Hello from interface B");
  }

} //is more specific because of the 'extend'

class C implements A, B { }

public class ThirdDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

Listing 2.41: Interface B is the most specific interface.

The output will be: "*Hello from interface B*".

Conflicts are not always avoidable

If we can't find an unique most specific default-providing interface, but we have more of them, an error will occur.

```java
interface A {
  default void m() {
    System.out.println("Hello from interface A");
  }
}

interface B {
  default void m() {
    System.out.println("Hello from interface B");
  }

}

class C implements A, B { }

public class FourthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

Listing 2.42: There is no more-specific interface between A and B.

The output will be: "*class C inherits unrelated defaults for m() from types A and B*

class C implements A, B ".

New sintax to resolve conflicts

To resolve a conflict, the method m must be implemented in the class that creates the conflict itself.

A special sintax X.super.m() can be used, where X is the superinterface that has the default method to be selected.

```
interface A {
  default void m() {
    System.out.println("Hello from interface A");
  }
}

interface B {
  default void m() {
    System.out.println("Hello from interface B");
  }
}

class C implements A, B {
  public void m() {
    A.super.m(); //calls m in A
  }
}
```

```
18
19 public class FifthDM {
20   public static void main(String[] args) {
21     C c = new C();
22     c.m();
23   }
24 }
```

Listing 2.43: The conflict is now resolved thanks to the call in line 15.

Remember: this new sintax is just for resolving conflicts while using default methods and not for a general purpose [Goe11].

About abstract methods

Again: classes always win. Here's a tricky example.

```
interface A {
  default void m() {
    System.out.println("Hello␣from␣interface␣A");
  }
}

abstract class B {
  abstract void m();

}

class C extends B implements A { }

public class SixthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

Listing 2.44: m() is considered abstract in C.

The output will be: "*C is not abstract and does not override abstract method m() in B class C extends B implements A { }*".

This happens because the abstract declaration in B takes precedence over the default declaration in A.

Extension methods are virtual methods

Just like the other methods, default methods are virtual in Java.

So we have to remember that the binding of methods is done dynamically at run-time.

```java
interface A {
  default void m() {
    System.out.println("Hello from interface A");
  }
}

interface B extends A {
  default void m() {
    System.out.println("Hello from interface B");
  }

}

class C implements B {}

public class SeventhDM {
  public static void main(String[] args) {
    A c = new C(); //Even if the static type of c is A,
    //method m in B will be executed
    //(because C is implementing B)
    c.m();
  }
}
```

Listing 2.45: B is the dinamic-type of the variable c.

The output will be: "*Hello from interface B*".

2.5.3 A real example in JDK 8

We know (from Section 1.2.2) that default methods have been introducted to make lambda-compatible the old libraries. The fundamental goal of backward compatibility is achived thanks to default methods, that make the old interfaces editable, supporting in this way the evolution of the libraries. For instance, see the code of the new java.lang.Iterable:

```
@FunctionalInterface
public interface Iterable<T> {
    Iterator<T> iterator();

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

Listing 2.46: The new forEach() method inside the Iterable<T> class.

Thanks to that, we can use functional interfaces and method references just as we learnt few paragraphs ago.

Print out a list has never been so easy:

```
List<String> list = . . .
list.forEach(System.out::println);
```

Listing 2.47: We are using the method reference to println.

2.5.4 Conseguences for the language

It is important to point out that introducing default methods is a great feature for developer but it is also a big step for the language.

The intent of this feature is to allow interfaces to be extended over time preserving backward compatibility. Developers maybe can choose to avoid the use of abstract classes at all and to use a lot this new feature, or maybe they will prefer to use default methods only in the few cases where there isn't a good alternative.

The idea behind an interface is to give a type, then, thanks to other features (like abstract class) we can build some kind of structure of the implementation. Default methods make interfaces more powerful, giving them the chance to be also a part of the implementation structure of a project.

Anyway, the bigger change we are looking at is the fact that now we have *inheritance of behavior* from multiple sources: and this is the topic of the next chapter.

Chapter 3

A Trait-Oriented approach

3.1 Introducing traits

A trait is a "*simple conceptual model for structuring object-oriented programs*" [DNS$^+$06]. To understand why traits are related to our previous discussions and to Java 8, we must briefly introduce this model. The purpose of this document is not to describe traits in their completeness (we suggest [DNS$^+$06, SDNB03] for a more detailed reading), but just to provide a path to put in contact traits and Java, thanks to the new features.

3.1.1 Why traits?

Composition and **decomposition** have always been two of the most important characteristic to care about in a programming language.
Let's point out some problems with inheritance and composability; then, we will describe traits and see how they can solve those problems.

- *Duplicated features.* Single inheritance is the basic form of inheritance; thanks to that we can reuse a whole class (and then, maybe, add some features). But sometimes we want to express something that is too much complex to be implemented with single inheritance. Just for example, we could have a class Swimming (that gives features for swimming animals) and a class Flying (that gives features for flying animals). What if we want to describe an animal that can both swim and fly? Poor swans! We can inherit only from Swimming or Flying but not both, so we will have to duplicate some of the existing features in the Swan class.

- *Inappropriate hierarchies.* Trying to avoid the above problem can bring us to this situation. Instead of duplicating methods in the lower-classes, we bring those methods up in the hierarchy and sometimes we can make the mistake of bringing them too much up, violating in this way the sense of the upper-classes.

- *Conflicting features.* If we have multiple-inheritance (as C++ does) a common problem is how to treat conflicts. Method conflicts can be solved thanks to overriding but conflicting attributes are more problematic. It is never clear if a conflicting actribute should be inherited once or twice and how these attributes should be initialized.

Traits are a possible solution to these (and others) problems.

3.1.2 What is a trait?

A trait, as proposed at first [DNS+06], is basically a collection of **methods**. This is very important: traits are statless, they cointain only methods (no fields) and so every state-conflict is automatically avoided (but of course we can have method conflicts, we are going to deep in this later).
Every trait can also define **required methods** and **required fields**. In our model, we will define required fields thanks to methods (so with setter and

getter).

A trait can be defined directly (by specifying its methods) or by composition from another trait (or more than one). The composition can be done thanks to operators (each one will be explained more completely later on):

- *Overriding*: a new trait is defined by adding method(s) to an existing trait.
- *Symmetric Sum*: a new trait is defined by putting together two or more existing traits.
- *Exclusion*: a new trait is defined by deleting a method from an existing trait.
- *Aliasing*: a new trait is defined by adding a second name to a method from an existing trait. In case of a recursive method is aliased, the recursive call will be done on the original method.
- *Duplicate*: it is like the aliasing but in case of a recursive method is duplicated, the recursive call will be done on the duplicated method.
- *Rename*: a new trait is defined by renaming every occurrence of a required/provided method.

Note that only overriding, symmetric sum and exclusion are from the original proposal [DNS$^+$06] while the others are from further works (for example [BDG08]). The original definition of traits says that classes and traits are well separated: the first ones are generator of instances, the second ones are units of reuse.

A class can be specified by composing a superclass with a set of traits and some *glue methods*. Glue methods are written inside a class and make possibile the connection between different traits. A perfect example of glue code

are the setter/getter, that give field access to the trait methods. Not only, glue methods are also used for resolve conflicts.
Trait composition respects the following three rules [DNS+06]:

- Methods defined in a class itself take precedence over methods provided by a trait. This allows glue methods defined in the class to override methods with the same name provided by the traits.
- Flattening property. A non-overridden method in a trait has the same semantics as if it were implemented directly in the class.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

A **conflict** happens when two traits specify an identical named method. Conflicts can be resolved thanks to the upon described operators: we could exclude one of the conflicting methods, we could use override, or maybe we can rename one (or all) of the methods involved in the conflict.

3.1.3 Traits versus mixins

Traits and mixins are related. They both have the goal of reuse. We outline the three main differences between traits and mixins:

- Mixins are stateful, that is, they can contain fields, while traditional traits don't.
- Mixins use "implicit conflict resolution" (the resolution is based on the semantic of the language) while traits use "explicit conflict resolution" (is all up to the developer).
- Mixins depend on linearization, traits are flattened.

About the second point, we recall that in mixins the application order defines the overriding order [FKF99] and the linking phase can do some automatic renaming [FF98]: this is out of the "all to the programmer" concept. Of course there are different implementation proposal of mixins, some of them totally rely on the semantic of the language for resolving conflicts, others give more low-level instruments to the developer so that he can guide part of the semantic.

If some of this concepts seem vague, don't worry, we will talk a lot about them.

3.2 From traits to a trait-oriented approach

Now we have an idea about what a trait is. We have understood that traits can be userful, and maybe some of you are thinking something like: "wouldn't it be nice if traits were implemented in Java?".

Traits have only been fully-implemented in Smalltalk and PHP 5.4 (and a few other languages) but a lot of literature proposes different models and implementation of them [DNS$^+$06, BDG08, SD05, BDNW08].

In this document we will not propose a Java implementation of traits but we will explore different solutions to program in a trait-oriented approach by taking advantage of the new Java 8 features. Finally, where it is possible, we will try to define some sort-of patterns to guide the developer through the use of this trait-oriented approach.

We will show how default methods can help us in this purpose but also the difference between a fully-implemented trait model and our approach.

Ultimately, our goal is to move the point of view from a pure single-inheritance approach to a more reusable way to write and use code.

3.2.1 Some introductions

First it is necessary to talk about Java 8 and how it can help/make difficult our goal to do some trait-oriented programming with it. SSome of the trait operators are easy to implement, others need workarounds (and sometimes the solutions we propose are not completely satisfactory).
Let's see which Java 8 elements are useful for our purpose and how:

- An interface plays the role of a trait. It is the first stone of our construction; interfaces can be traits thanks to the new default methods. Moreover, we can implement multiple interfaces and this corresponds to the composability of traits (we must be able to sum more than two traits simultaneously).

- Most of the trait operators permit the resolution of conflicts simply by working on the set of method names of the involved traits. However, the trait override works at a lower level, that is, implies writing some code in order to obtain the sought effect. Within Java 8, we have the X.super.m() functionality, which is useful to select the implementation we want by indicating explicitly the trait. This mechanism is also reminiscent of some mixin implementation, as pointed out in Section 3.1.3.

Finally let's see a first example of an interface/trait. From now on, we will refer at interfaces used like traits simply as traits.

```
public interface TOccurrenceCounter {

  public String getString();

  /**
   * Counts the occourrences of str inside the string field.
   */
  default int count(String str) {
        String field = getString();
        Int counter = 0;
        // ...
        return counter;
    }

}
```

Listing 3.1: An example of a trait.

As we can see, this traits provides a simple functionality: it searches the occurrences of a string str inside a field (that is required by the required method getString()).
We will use this formalism: a trait starts its name with a T or with Trait.

In the next sections we provide a pattern for each of the previous described operators, then, at the end, we will see a more complex example where the utility of this approach will come up.

3.2.2 Operator: Trait Override

Let's deal with the first operator: *trait override*. We are lucky! Java helps us a lot in this case and so we have a 'free' operator.
The override operator defines a new trait by adding one or more methods to an existing one. To obtain this effect, we can simply use the interface extension we well know.

```
public interface TraitA{
  default void m() {
    System.out.println("Hi from m, in TraitA");
  }

}
```

Listing 3.2: A simple trait with a method.

Now, we can override TraitA by creating a new interface extending it, and just write down the new method(s) we want to add.

```java
public interface TraitB extends TraitA {

  /** Overrides TraitA, adding a new feature **/
  public void m2() {
    System.out.println("Hi from m2, in TraitB");
  }

}
```

Listing 3.3: TraitB overrides TraitA.

It seems to work well. Just to be sure, let's create a little main to test it out.

```java
public class C implements TraitB {

  public static void main(String[] args) {
    C c = new C();
    c.m();
    c.m2();
  }
}
```

Listing 3.4: Both methods (m() and m2()) are callable from a C object.

It is impossibile to have a conflict: if the overriding method is already defined in the upper-trait, the new implementation is the one that will be executed once the method is called.

3.2.3 Operator: Symmetric Sum

This is one of the most important operators as it provides the fundamental feature of multi-inheritance. In a single trait we can commixture more capabilities from different traits. Let's spend a few words on the *symmetric* adjective. With symmetric we mean that all the addends are "peers": there is no overthrow from one or more of the addends. This means that in case of conflict it is up to the developer to deal with it because there is no semantic reason to put a trait over the others.

Let's go back to our operator. We will have to distinguish two different cases.

Symmetic Sum without conflicts. Let's consider three simple traits: TMouth, TEyes and TTail. Each trait defines some actions that are possibile thanks to a certain animal's body part.

```
public interface TMouth {
  default void makeASound() {
    System.out.println("I'm making a sound");
  }

  default void eat(String s) {
    System.out.println("I'm eating "+s+".");
  }
}
```

Listing 3.5: The TMouth trait.

```
public interface TEyes {
  default void lookAround() {
    System.out.println("I'm looking around...");
  }

  default void blink() {
    System.out.println("I'm blinking");
  }

}
```

Listing 3.6: The TEyes trait.

```
public interface TTail {
  default void shakeTail() {
    System.out.println("Wuush, Wuush, I'm shaking my tail.");
  }
}
```

Listing 3.7: The TTail trait.

Our purpose is to have a new trait, TCat, that puts together all the features from every single body part previously defined. We can create a fresh new trait defined in this way:

```
public interface TCat extends TEyes, TMouth, TTail {
  default public void purr(){
    System.out.println("PuUurRrRr");
  }
}
```

Listing 3.8: The TCat trait is the sum of three other traits.

The sum is applied by extending simultaneously multiple traits. To make it more interesting, we also used trait override by adding a purr() method, but this has nothing to do with the sum itself.

In our test class, we can see that every method from each of the initial trait it is callable.

```
public class MyCat implements TCat {

  private String name;

  /**
  * Constructor
  */
  public MyCat(String n) {
    this.name = n;
  }

  public static void main(String[] args) {
    MyCat jacky = new MyCat("Jacky");
    jacky.eat("Meat");
  }
}
```

Listing 3.9: For instance we tried the eat(String) method from TMouth.

It worked kinda smoothly, didn't it? We were lucky in this case because every trait involved in the sum was **disjoint** from the others. In other words, we didn't have a conflict because the traits we were summing didn't have any common named method.

Symmetic Sum with conflicts. If we introduce a method close() in both TMouth and TEyes (meaning that the cat could close the mouth or close the eyes), then when compiling MyCat (Listing 3.9) we will get this error:

MyCat.java:1: error: class MyCat inherits unrelated defaults for close() from types TEyes and TMouth public class MyCat implements TCat {}.

We had a conflict. To solve this conflict we have different options. One of them is to use the keyword X.super.m(). So, we can edit our TCat trait in this way to fix the problem:

```java
public interface TCat extends TEyes, TMouth, TTail {

  /** Conflict resolution **/
  default void close() {
    TEyes.super.close();
  }

  default public void purr(){
    System.out.println("PuUurRrRr");
  }
}
```

Listing 3.10: One out of the two close() methods is selected.

The method close() that will be executed is the one from the TEyes trait. This is not the most suitable solution, as the close() method from the TMouth trait is completely lost, but is still a good way to resolve a conflict (supposing that we do not need one of the two involved methods). The use of the X.super.m() feature reduces the low coupling between TCat and TEyes: if someday the close() method will change (for example by adding a parameter to it), also the close() method inside TCat has to change (and this is bad!).

3.2.4 Operator: Alias

The alias operator provides another name for referring a certain method. Again, it is a quite simple task.

Consider this simple trait:

```java
public interface TraitA {
  default void mOneA() {
    System.out.println("Hi from method one in A.");
  }

  default void mTwoA() {
    System.out.println("Hi from method two in A.");
  }
}
```

Listing 3.11: A trait providing two methods.

Now, in a new trait (TraitB) we create an alias for the mTwoA().

```
public interface TraitB extends TraitA {

  /** Aliasing mTwoA() in aliasMTwoA() **/
  default public void aliasMTwoA() {
    mTwoA();
  }
}
```

Listing 3.12: mTwoA() has now an alias called aliasMTwoA().

And then, as usual, a test class:

```
public class MyB implements TraitB {
  public static void main(String[] args) {
    MyB mc = new MyB();

    mc.aliasMTwoA();
    mc.mTwoA();
  }
}
```

Listing 3.13: We can refer to the same method with both names.

When you apply the alias operator, is important take care of the alias name: you may by mistake ovveride another method from the upper-trait.

3.2.5 Operator: Duplicate

This operator is just the same as the alias but in case of recursive method, the recursion-call will refer to the duplicated method instead that to the original one. To point out this peculiar characteristic, we will see an example with a recursive method.

Consider this trait:

```
public interface TraitA {

  default String count(int c) {
    if(c == 0) return "\n- Finally reached 0.\n";
    else return "\n- This laps is "+ c + count(c-1);
  }

}
```

Listing 3.14: The count() method is recursive.

Nothing cloudy here, it is just a method that counts its own recursive-calls and builds up a string to illustrate them.

Suppose that we are interested in duplicating the count() method, we can create a new trait and make the duplication inside of it. Here's how to do it:

```
public interface TraitB extends TraitA{

  /** duplicate of counter **/

  default String count_dup(int c) {
    return TraitA.super.count(c) + "-Knock-";
  }

  /** If you comment this method,
    we have a simple aliasing  **/
  default String count(int c) {
    return count_dup(c);
  }
}
```

Listing 3.15: The count() method is now duplicated as count_dup().

How it works? The count_dup() does basically an aliasing. But then we also add a count() method that overrides the TraitA's count(), so, in this way, every recursive call will be sent to the method inside of TraitB instead of the one in TraitA. The overrding count() bounces back the call to the count_dup() and so we have the desired effect.

Let's try it out:

```
public class C implements TraitB{

  public static void main(String[] args){
    C c = new C();
    System.out.println(c.count_dup(5));
  }

}
```

Listing 3.16: We call the duplicated method.

In C class we can see that the duplicate method works perfectly, as the output is the following:

```
- This laps is 5
- This laps is 4
- This laps is 3
- This laps is 2
- This laps is 1
- Finally reached 0.
-Knock--Knock--Knock--Knock--Knock--Knock-
```

Listing 3.17: The output from the C execution.

The long knock-list (a '*-Knock-*' is concatenated every time count_dup() is called) tells us that everything is working good.

Some reflections:

- If count() wasn't a recursive method, the effect would be **exactly the same** as if we used aliasing.
- Calling count() or count_dup() gives exactly the same result, but if we call count_dup() instead of count() we could save one method-call on the stack.

3.2.6 Operator: Exclude

This is a tough operator. Is well known that once you add something in your code, is hard to remove it.

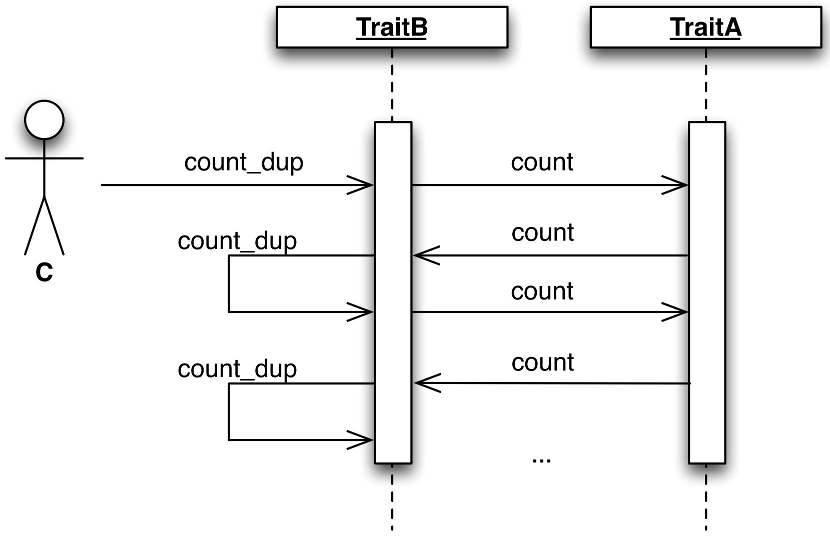

Figure 3.1: A visual rappresentation of the recursive calls.

Initially, in [Goe11], it was described the possibility to remove a default method by using the *default none* keyword. We didn't find any other trace of that operator on any other further document: we can assume that it is not avaliable in Java 8.

This decision is understandable because of the fact that this operator would add an extra hard work on the compilator: processing negative information is never easy.

The default none operator would have been perfect for our purpose: sadly, we weren't able to obtain the same results as if it was still available. Let's explore our solution and then we will highline the drawbacks. Consider this trait:

```
public interface TraitA {
  default void m() {
    System.out.println("Hi from m, in TraitA");
  }

  default void q() {
    System.out.println("Hi from q, in TraitA");
  }
}
```

Listing 3.18: A simple trait that defines two methods.

If we want to delete one of the two methods (let's say m()), we can do it in a new trait in this way:

```
public interface TraitB extends TraitA {
  /* It's an empty implementation, it means exclusion. */
  default void m() { }
}
```

Listing 3.19: We ovveride m() with an empty body.

We do not delete the method for real, we just blur the upper-implementation with an empty method.

Here's our usual test-class:

```
public class C implements TraitB {

  public static void main(String[] args){
    C c = new C();
    c.m();
    c.q();
  }
}
```

Listing 3.20: m() is still callable but does nothing at all.

Let's make some observations:

- The upper-implementation will never be available again in the below classes/traits (this is ok, it was one of our purposes).
- It is important understand that the excluded method isn't deleted at all. This means that if we use methods like C.class.getMethod('m'), we will still get a response! It is something that the developer has to keep in mind.

- If we are doing a trait-sum and in both traits we have a method that we want to exclude, then this operator will work perfectly and will exclude simultaneously both the upper-methods.

We didn't get a really satisfying solution here. However we are just trying to provide a trait-oriented approach doing the best we can with Java 8 features.

3.2.7 Operator: Rename

Last but not least: rename. Technically the rename operator can be applied to provided methods and required methods.
We provide two different approaches for each of the two cases.

Renaming a required method

We have two possibile implementations: each one has certain advantages and drawbacks.

Using an interface. Let's think as we did before. We take an interface that asks for a required method (corresponding to a signature without a body).

```java
public interface TraitA {
  public String neededMethod(int par);

  default String providedMethod(int ext_par) {
    return "My needed method says: " + neededMethod(ext_par);
  }
}
```

Listing 3.21: requiredMethod() has to be implemented.

Then, we define another trait in this way:

```java
public interface TraitB extends TraitA {
  public String renamedNeededMethod(int par);

  /**
```

```
   * Makes the renaming possibile.
  **/
  default String neededMethod(int par) {
    return renamedNeededMethod(par);
  }
}
```

Listing 3.22: requiredMethod() is renamed as renamedNeededMethod().

The idea is to provide the TraitA's required method and to require another one from TraitB. Then, we just put the right calls in the right places.

Let's see our test class:

```
public class C implements TraitB {

  public String renamedNeededMethod(int par) {
    return "Impl. of renamedNeededMethod in C, par = " + par;
  }

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.providedMethod(6));
  }
}
```

Listing 3.23: C is providing renamedNeededMethod().

Everything works smoothly but we have an enormous drawback: if any class implementing TraitB will ever override neededMethod() (and this is possible because it is a default method just like the others), our renaming is blown-up.

How can we protect a default method from being overridden? It is kinda of a non-sense, as a default method is by definition easy to override. In fact, we can't use the final modifier while declarating a default method.

Using an abstract class. Given the problem, we can have another approach by using an **abstract class**. In this way, the final modifier could be used and we could protect our neededMethod(). Let's see the code.

```java
public abstract class TraitB implements TraitA {
  public abstract String renamedNeededMethod(int par);

  /** Rename **/
  public final String neededMethod(int par) {
    return renamedNeededMethod(par);
  }
}
```

Listing 3.24: TraitB is now an abstract class.

And this is the C class:

```java
public class C extends TraitB {

  public String renamedNeededMethod(int par) {
    return "Impl. of renamedNeededMethod in C, parameter: "+par;
  }

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.providedMethod(6));


  }
}
```

Listing 3.25: Now C extends TraitB because TraitB is a class.

Now, if we add a neededMethod() inside C we get this error:

```
.java:7: error: neededMethod(int) in C cannot
          override neededMethod(int) in TraitB
  public String neededMethod(int p) {
                ^
  overridden method is final
```

Listing 3.26: We can't override neededMethod() anymore.

This solution works better but has also a big drawback: TraitB isn't an interface anymore and this breaks our interfaces-traits concept. Also, every class that is interested in the use of TraitB has to extend it (just like C did) and as we have single inheritance, it will be impossible to extend any other class.

Renaming a provided method

Renaming a provided method is by far the common way to resolve a conflict. So we will firstly see how to rename a provided method, then, we will use this feature to resolve a conflict.

We have to think to the rename operator as a combination of two previous operators: **rename** = **exclude** *plus* **alias**.

Let's see how it works. We have our usual simple trait:

```java
public interface TraitA {

  default void m() {
    System.out.println("Hi from m() in TraitA!");
, }
}
```

Listing 3.27: A simple trait.

Then, we rename m() as mTraitA() thanks to this code in TraitB:

```java
public interface TraitB extends TraitA {

  /* Exclude */
  default void m() {}

  /* Aliasing */
  default void mTraitA() {
    TraitA.super.m();
  }

}
```

Listing 3.28: We see the combined use of aliasing and exclude.

This approach works well, but brings the inevitable well known drawbacks from the exclude operator. Anyway, we will see that during a conflict-resolution this won't be a problem. As we can see, the fresh mTraitA() can be called inside our test-class:

```java
public class C implements TraitB {}

```

```
  public static void main(String[] args) {
    C c = new C();
    c.mTraitA();
  }
}
```

Listing 3.29: The new name is callable.

Note.: This approach (and the following) couldn't work for renaming a required method, as the X.super.m() would not compile (it would refer to an abstract method!).

Resolving conflicts with rename. When we deal with a conflict, we can:

- Rename all the conflicting methods: this approach is very similar to the one described above.
- Rename all the conflicting methods but one: in this way, we can solve the drawback from the exclude operator.

Let's see the two approaches apart.

Renaming all the conflicting methods. Renaming all conflicting methods could be a nice idea to have clearer names.
We have two traits we have alredy seen before: TMouth and TEyes both defining a close() method.

```
public interface TMouth {
  default void makeASound() {
    System.out.println("I'm␣making␣a␣sound");
  }

  default void eat(String s) {
    System.out.println("I'm␣eating␣"+s+".");
  }

  default void close() {
    System.out.println("I'm closing my mouth");
  }
}
```

Listing 3.30: TMouth defines a close() method.

```
public interface TEyes {
  default void lookAround() {
    System.out.println("I'm looking around...");
  }

  default void blink() {
    System.out.println("I'm blinking");
  }

  default void close() {
    System.out.println("I'm closing my eyes");
  }
}
```

Listing 3.31: TEyes also defines a close() method.

Now, we put them together in the TCat trait:

```
public interface TCat extends TEyes, TMouth, TTail {

  /** Conflict resolution **/
  default void close() {}

  /** First renaming **/
  default void closeEyes() {
    TEyes.super.close();
  }

  /** Second renaming **/
  default void closeMouth() {
    TMouth.super.close();
  }

  default public void purr(){
    System.out.println("PuUurRrRr");
  }
}
```

Listing 3.32: We resolve the conflict by renaming all the conflicting methods.

Thanks to this approach, the two close()'s are much more clear and so the code is better documented. Remember: we still have the drawback from the exclude operator.

Renaming all the conflicting methods but one. In the above example, we left totally unused the close() method. Why don't we use it for calling one of the conflicting methods?

```java
public interface TCat extends TEyes, TMouth, TTail {

  /** Conflict resolution and first renaming **/
  default void close() {
    TEyes.super.close();
  }

  /** Second renaming **/
  default void closeMouth() {
    TMouth.super.close();
  }

  default public void purr(){
    System.out.println("PuUurRrRr");
  }
}
```

Listing 3.33: The exclude drawback is resolved.

One of the conflicting methods (TEyes's one) uses now the old conflicting name, while the other(s) are renamed.
The exclude drawback is now resolved because there is no empty-method (the one that should be empty is close(), instead is used to provide one of the conflicting methods implementation).

3.3 A common problem

While programming in a trait-oriented approach it is important to pay a particular attention to the override (the Java override, not the trait override). Lot of our operators relate to Java override and sometimes that can substantially blow up all our work by blocking us in the use of rename and exclude operators (as the first is based on the second).

We have seen that is extremely easy generate new code, new names, but it is very hard to delete something that already exists. This is the real problem: when we rename or we exclude a method, the old name that the method used doesn't become avaliable again, it is, instead, forever binded to his first definition (in terms of return type).

This makes the reuse of that name particullary fragile, and, sometimes, impossibile. To present this problem we will consider a simple case study. It is the first time that our operators will be working on something bigger than a built-up example.

3.3.1 The stack example: an old approach

Before starting to dig into the described problem, we will illustrate how our example can be developed in the "classical single inheritance style", so, later on, we will be able to make some comparations between the approaches, and to see (despite the implementation problems) how the trait one can be more convenient.

This example is inspired from [BDG08]. We want to develop a stack data structure. We can create an interface IStack, then, a class Stack that implements it and has a main just to make sure that everything works correctly. Here's the IStack interface:

```java
public interface IStack {

  /* Tells if the stack is empty */
  public boolean isEmpty();

  /* Adds one item on the stack */
  public void push(Object obj);

  /* Removes and returns the first object on the stack */
  public Object pop();
}
```

Listing 3.34: The interface for our stack.

We have the implementation of this interface, as follows:

```
import java.util.*;

public class Stack implements IStack {
  LinkedList<Object> l;

  public Stack() {
    l = new LinkedList<Object>();
  }

  public boolean isEmpty() {
    return l.isEmpty();
  }

  public void push(Object obj) {
    l.add(obj);
  }

  public Object pop() {
    return l.removeFirst();
  }

  public static void main(String[] args) {
    Stack st = new Stack();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
    st.push("Hello");
    System.out.println("Just popped: "+st.pop());
    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
  }
}
```

Listing 3.35: The implementation for the IStack interface.

Now, suppose that we have to implement another stack, with this interface:

```
public interface IStackAlt {
  public boolean isEmpty();
  public void push(Object obj);

  /* Removes the first object on the stack */
  public void pop();

```

```java
  /* Returns the first object on the stack (not removing it) */
  public Object getTop();
}
```

Listing 3.36: Another interface for our stack.

As we can see, this interface is different from IStack because of two methods: pop() is now void, and we have an additional method getTop(). Again, we can develop the implementation for this interface as follows:

```java
import java.util.*;

public class StackAlt implements IStackAlt {
  LinkedList<Object> l;

  public StackAlt() {
    l = new LinkedList<Object>();
  }

  public boolean isEmpty() {
    return l.isEmpty();
  }

  public void push(Object obj) {
    l.add(obj);
  }

  public void pop() {
    l.removeFirst();
  }

  public Object getTop() {
    return l.getFirst();
  }

  public static void main(String[] args) {
    StackAlt st = new StackAlt();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
    st.push("Hello");

    System.out.println("What's the first element? "+st.getTop());
    System.out.println("I'll pop it out!");
    st.pop();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
  }
}
```

Listing 3.37: The implementation for the IStackAlt interface.

We can make a simple observation: both methods isEmpty() and push() were already implemented inside the Stack class and we had to re-implement them inside the StackAlt class! This is a waste, isn't it?
Let's see what we can do thanks to our trait-oriented approach.

3.3.2 The stack example: a trait-oriented approach

Let's try to use our trait-approach to promote the code reuse. Do not forget that our purpose here is to illustrate the difficulty to work with Java override, so we will show two different situations where we can encounter problems related to that.

Renaming a non-void method into a void one

First of all, we can introduce a TStack trait that defines all the necessary features to work with a stack.

```java
import java.util.*;

public interface TStack {
  public List<Object> getStructure();

  default boolean isEmpty() {
    return getStructure().isEmpty();
  }

  default void push(Object obj) {
    getStructure().add(obj);
  }

  default Object pop() {
    Object o = getStructure().get(0);
    getStructure().remove(0);
    return o;
  }

}
```

Listing 3.38: The TStack trait.

We can see the use of a new method, getStructure() to have access to the structure that will effectively implement the stack. Then, we have the implementation:

```
import java.util.*;

public class Stack implements TStack {
  LinkedList<Object> l;


  /* Glue Code */
  public LinkedList<Object> getStructure() {
    return l;
  }

  public Stack() {
    l = new LinkedList<Object>();
  }


  public static void main(String[] args) {
    Stack st = new Stack();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));

    st.push("Hello");

    System.out.println("Just popped: "+st.pop());

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
  }
}
```

Listing 3.39: The Stack class using the TStack trait.

This works pretty well. We introduced some glue code to provide the previously mentioned getStructure() method. Now, we want introduce a new method getTop() and we want to change the old pop() that was returning an Object into a void version. The first goal is easy, we can use the override operator. But when we try to deal with the pop() method, we encounter this problem:

```
public interface TStackAlt extends TStack {

  /** We redefine pop with a void return type **/
  default Object pop() {
    getStructure().remove(0);
    return null;
  }

  /**
    If we could, we would have done a simple renaming:

    default Object pop() { return null; }

    default void pop() {
      getStructure().remove(0);
    }

    But we cannot override a method with different
    return types. So we mixed this method and the
    excluding one up here.
  **/


  /** We make the old pop stil avaliable (optional) **/
  default Object popTop() {
    return TStack.super.pop();
  }

  /** Override **/
  default Object getTop() {
    return getStructure().get(0);
  }

}
```

Listing 3.40: The TStackAlt trait extends the TStack one.

As the comment says, it is impossibile ovveride the pop() method as the return type (void) is different from the previously defined (Object).

Why this happens? We already explained that the the renaming is an exclusion plus an aliasing: but our exclusion it is just a weak imitation of a real exclusion, we do not exclude the method at all.

The problem isn't inside the Java ovveride, but is in the use we make of this feature. This problem happens because our intent is to reuse the already assigned pop() name. Again: when we deal with an already defined method we have to pay a lot of attention.

Notice that this isn't really a problem of our trait-approach, in fact we would have the exact same problem even if we were programming in a single inheritance approach (because this is how the Java override works).

In this scenario we were although lucky! We did, infact, provide a (bit dirty) solution. Returning a null value fixs the problem. Let's see our implementing class:

```
import java.util.*;

public class StackAlt implements TStackAlt {
  LinkedList<Object> l;

  public StackAlt() {
    l = new LinkedList<Object>();
  }

  /* Glue Code */
  public LinkedList<Object> getStructure() {
    return l;
  }

  public static void main(String[] args) {
    StackAlt st = new StackAlt();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));

    st.push("Hello");

    System.out.println("What's the first element? "+st.getTop());
    System.out.println("I'll pop it out!");
    st.pop();

    System.out.println("Is empty? "+(st.isEmpty()? "Yes":"No"));
  }
}
```

Listing 3.41: The main is exactly the same as in the old approach.

Again, we have some glue code. But we can see that the main have no difference from the one in the old approach (Listing 3.36). This is a success! All considered, we had an hitch but we came out of it pretty well. The two isEmpty() and push() methods aren't repeated anymore (good for code reuse), the class tree is more clear, we provided a new pop() method but we also made the old one still accessible and we didn't have to change a single line inside the main (and this is good if we suppose that it isn't our code but it comes from somewhere else).

Renaming a void method into a non-void one

As told before, we were lucky. If we were in the opposite case we wouldn't be able to find a good solution. Let's invert the situation: now TStack defines a void pop() and TStackAlt wants to define a new Object-returning version of it.

We have the two traits defined as follows:

```java
import java.util.*;

public interface TStack {
  public List<Object> getStructure();

  default boolean isEmpty() {
    return getStructure().isEmpty();
  }

  default void push(Object obj) {
    getStructure().add(obj);
  }

  default void pop() {
    getStructure().remove(0);
  }

  default Object getTop() {
    return getStructure().get(0);
  }

}
```

Listing 3.42: The TStack has now a void pop().

```
public interface TStackAlt extends TStack {

  /**
    We CAN'T redefine this method,
    as Object is a different return type
    from void.

    default Object pop() {
      Object o = getStructure().get(0);
      getStructure().remove(0);
      return o;
    }
  **/

  /**
    We make an alternative pop,
    but it has another name.
    This means that the main has to change.
  **/
  default Object altPop() {
    Object o = getTop();
    pop();
    return o;
  }


}
```

Listing 3.43: The TStackAlt wants to define an Object-returning pop().

We just can't override a method that is void by changing his return type in something that is not void. All we can do, is to define another pop() method that finally returns an Object and call it inside our main. This means that the main must change and this could be not suitable/possibile in certain cases. Let's see how the StackAlt has changed.

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

```

public class StackAlt implements TStackAlt {
  LinkedList<Object> l;

  public StackAlt() {
    l = new LinkedList<Object>();
  }

  /* Glue Code */
  public LinkedList<Object> getStructure() {
    return l;
  }

  public static void main(String[] args) {
    StackAlt st = new StackAlt();

    System.out.println("Is␣empty?␣"+(st.isEmpty()? "Yes":"No"));

    st.push("Hello");

    System.out.println("Just␣popped:␣"+st.altPop());

    System.out.println("Is␣empty?␣"+(st.isEmpty()? "Yes":"No"));
  }
}
```

Listing 3.44: We had to change our main by using altPop() instead of pop().

The call to pop() is now changed into a call to altPop().

This solution is perfectly working if we can edit the main, but if that code isn't editable then we are just stucked.

The covariant override

Just to be complete about this argument, the override in java is a **covariant override**: an overriding method may have a result type that is a subtype of the method it overrides.

This means that it is legal (while you work with your operators) to override a method from an upper trait by keeping the same return type or a subclass of it, but the opposite is illegal. So we may do something like this:

```java
public interface TraitA {

  default Object m() {
    // something
  }

}

————————————————————————————————————————————————————————————————

public interface TraitB extends TraitA {

  default String m() {
    // something else
  }

}
```

Listing 3.45: This override is legal.

This works because String is a subtype of Object. We can't do the opposite:

```java
public interface TraitA {

  default String m() {
    // something
  }

}

————————————————————————————————————————————————————————————————

public interface TraitB extends TraitA {

  default Object m() {
    // something else
  }

}
```

Listing 3.46: This override isn't legal.

This doesn't work because Object isn't a subtype of String.

Mind that if you are working with primitive types, you will have to use the wrapper version to make the covariant override work.

3.4 A summary

We introduced a lot of information, and maybe is a good idea to zoom-out a little bit. Let's get a brief summary of all the things we have learnt.

– We can think of *interfaces as traits*, where the provided methods are all defaults. If we need to define a required method we can add an abstract method and this is the only way for dealing with fields and their accessory methods.

– We can introduce multiple traits by implementing multiple interfaces.

– If a trait defines required methods then we will have to write some *glue code* (either in another trait or in a class) to make all the things work together.

– To add a method to a trait, we can use *trait override* (Section 3.2.2).

– To put together two or more traits, we can use the *symmetric sum* (Section 3.2.3) minding to check for any conflict.

– If we need another name for a method, we can use *alias* (Section 3.2.4) or *duplicate* (Section 3.2.5): mind that if we have a recursive call, the first operator will always call the original method, while the second will call the duplicated method.

– If we want to delete a method, we can use *exclude* (Section 3.2.6): it has a drawback, the method isn't really deleted but we just re-implement it with an empty body (to blur the upper-implementation).

– *Rename* (Section 3.2.7) works differently if we want to rename a provided method or a required method:

  - Renaming a required method (Section 3.2.7) can be done in two different ways, in the first we have the drawback of not being able to override the original method anymore, in the second we have the drawback that the trait is now a class and not an interface anymore.
  - Renaming a provided method (Section 3.2.7) uses the rename operator as *exclude plus alias*.

- A conflict happens when two methods have the same name. We can resolve conflicts:
  - Thanks to the `X.super.m()` (Section 3.2.3), minding that it has the drawback of breaking the low-coupling principle.
  - By renaming a provided method (or more) that is involved in the conflict. It is possibile to choose to reuse the conflictual name to avoid the exclusion drawback (Section 3.2.7).
- Always remember that while you are using your operators, you may encounter some of the overriding problems outlined in Section 3.3.

3.5 A bigger example

It is time to see a bigger study case where we can observe the whole mechanism working together.

We have the task to develop some features of a game. The player has a character that is able to move from one game-place to another. The goal of the game is to collect as many coins as possible (we will represent them with integers). Sometimes, during the game, the character can find himself in a room that has three doors (font-left-right). Every door can have different features and so provide a different amount of coins. A door can be opened only if it is unlocked (we will use a boolean value for that).

Our task is to develop a particular room with three different doors (mind that each one has different features).

The goal of code reuse is in this case very important because:

1. We have a lot of different rooms during the game, composed by different doors (but a kind of door can be used more than once in different rooms).

2. The special features binded with the doors could be used somewhere else during the game (in different situations).

Let's build some code!

3.5.1 The first concept: doors

The center of our task is the development of three doors (each one with different features), so, let's make a "base-door":

```java
import java.lang.Math;

/* Defines a base-door with no particular features */
public interface TDoor {

  /* State references */
  public boolean getLocked();

  /* Coin management */
  public int getDoorMaxCoins();

  /* Tells if the door is locked or not */
  default boolean isLocked() {
    return getLocked();
  }

  /* Tries to open the door */
  default int open() {
    if(!isLocked()) {
      System.out.println("The door has been opened!");
      int coins = (int) (Math.random() * getDoorMaxCoins()) + 1;
      System.out.println("You got "+ coins +" coins.");
      return coins;
    } else {
      System.out.println("This door is locked.");
      return -1;
    }
  }

  /* Performs a knock on the door */
  default int knock() {
    System.out.print("Door says: ");
    System.out.println("How you dare, I am the one who knocks!");
    int c = (Math.random()<0.8)? 0 : 1;
    if(c > 0)
      System.out.println("Ow! You got a free coin!");
    return c;
  }
}
```

Listing 3.47: A basic definition for our door concept.

As we can see, it is possibile open the door (only if it is not locked). Once opened, the door gives some coins (at least one, so the negative return works good to say "the door is still closed"). As additional action, is possibile to knock the door and try to get some free coins (there's a low percent for this event to happen, and anyway you can get max 3 coins).

3.5.2 The features

Now, we will develop three a-part features (we can imagine a lot of them). Notice that those features aren't related to the doors, so their code can be used everywhere in the game.

We have a *counter* that after a certain limit releases coins.

```
import java.lang.Math;

/* Provides a counter that after a limit releases coins */
public interface TCounter {

  /* State references */
  public int getCounter();
  public void setCounter(int c);
  public int getLimit();
  /* Coin management */
  public int getCounterMaxCoins();

  default void incrementCounter() {
    setCounter(getCounter()+1);
  }

  default void decrementCounter() {
    setCounter(getCounter()-1);
  }

  default boolean hasReachedLimit() {
    if(getCounter() >= getLimit())
      return true;
    else
      return false;
  }

  default int releaseCoins() {
    int co = (int) (Math.random() * getCounterMaxCoins()) + 1;
    System.out.println("You got "+co+" coins.");
    return co;
  }
}
```

Listing 3.48: The TCounter trait.

This is just a counter, it could work everywhere inside our game.

Then, we develop a *chest* that cointains coins (from 0 to a certain number).

```
import java.lang.Math;

/* Provides a chest that contains coins */
public interface TChest {

  /* Coin management */
  public int getChestMaxCoins();

  /* Opens the chest */
  default int open() {
    System.out.print("The chest is now opened!");
    int c = (int) (Math.random() * getChestMaxCoins());
    System.out.println(" You got "+ c +" coins from the chest.");
    return c;
  }
}
```

Listing 3.49: The TChest trait.

Another feature is the *enchantment*. An enchantment can both give or remove coins (the choice is based on a probability).

```
import java.lang.Math;

/* Provides an enchantment that can give or take coins */
public interface TEnchantment {

  /* Coin management */
  public int getEnchantMaxCoins();


  /*
    An enchantment can give coins  (max getEnchantMaxCoins())
    or remove coins (max -getEnchantMaxCoins())
  */
  default int applyEnchantment() {
    System.out.println("\nThis is an enchantment!");
    System.out.print("\"If the luck is up, ");
    System.out.println("of coins you'll have a cup,");
    System.out.print("but if no luck you got, ");
    System.out.println("you are gonna lose a lot.\"");

    int max = getEnchantMaxCoins();

    int coins = - max + (int) (Math.random() * ((max * 2) + 1));

    if(coins >= 0)
      System.out.println("Ohoh! You got "+coins+" coins!");
    else
      System.out.println("You lost "+Math.abs(coins)+" coins!");

    return coins;
  }
}
```

Listing 3.50: The TEnchantment trait.

It is natural see how a Door can simply become a ChestDoor or maybe an EnchantmentDoor. It could be easy also create a ChestEnchantedDoor! This is the big advantage of traits: it is easy compose them to create whatever we need.

3.5.3 The coin management

Some of you may have noticed in the previous traits a few particular required methods such as getEnchantMaxCoins() and getDoorMaxCoins(). It is a development choise to let the class using the trait define the max amount of coins that every element can release.

3.5.4 Composing the doors

Let's compose our TDoor with different features to obtain different featured-doors. We will make a trait that commixtures a door with a feature, then, we will create a class implementing it (and providing the glue code).

The EnchantedDoor

Let's make our TEnchantedDoor trait.

```java
/* Puts together a door and an enchantment */
public interface TEnchantedDoor extends TDoor, TEnchantment {

  /**
    When you open an enchanted door,
    you break the enchant and so you
    apply it.
  **/
  default int open() {
    int coins = TDoor.super.open();
    if(coins > 0) { //if the door is open
      coins += applyEnchantment();
    }
    return coins;
  }


}
```

Listing 3.51: We override the open() method from TDoor.

The enchant is broken only when the door gets opened. Let's make the actual class that puts everything together.

```java
public class EnchantedDoor implements TEnchantedDoor {

  /* Fields for the door */
  private boolean locked;

  /* Glue Code for TDoor */
  public boolean getLocked() { return this.locked; }

  /* Glue code for coin management */
  public int getDoorMaxCoins() { return 120; }

  public int getEnchantMaxCoins() { return 150; }

    /* Constructor */
  public EnchantedDoor(boolean l) {
    setLocked(l);
  }

  /* Other helpful methods */
  public void setLocked(boolean l) { this.locked = l; }

}
```

Listing 3.52: This is our final-working EnchantedDoor object.

The class using the trait defines the methods required by the trait and gets all the other methods provided from it. It is an incremental way of create objects.

Just to make it clear: in TEnchantedDoor we managed the fusion between two traits, and in EnchantedDoor we gave the required glue code and we made an actual instantiable class.

The ChestedDoor

Just like before, we create a trait that fuses together TDoor and TChest.

```java
/* Puts together a door and a chest */
public interface TChestedDoor extends TDoor, TChest {

  /**
    When you open a chested door,
    you also get the prize from the chest.
    This overrides the TDoor's open().
  **/
  default int open() {
    int coins = TDoor.super.open();
    if(coins > 0) { //if the door is open
      coins += openChest();
    }
    return coins;
  }

  /* Rename of open() from TChest */
  default int openChest() {
    return TChest.super.open();
  }




}
```

Listing 3.53: The TChestedDoor resolves a conflict by using the rename.

We did a few things all in once. We had to manage a conflict between the two open()'s methods (one from TDoor and the other from TChest). We renamed the open() from the TChest trait as openChest(). Then, we used the technique explained in Section 3.2.7 to reuse the conflicting name for re-implement the TDoor's open() (like we already did for TEnchantedDoor). Let's make the class that implements our trait:

```java
public class ChestedDoor implements TChestedDoor {

  /* Fields for the door */
  private boolean locked;

  /* Glue Code for TDoor */
  public boolean getLocked() { return this.locked; }

  /* Glue code for coin management */
  public int getDoorMaxCoins() { return 120; }

  public int getChestMaxCoins() { return 250; }

  /* Constructor */
  public ChestedDoor(boolean l) {
    setLocked(l);
  }

  /* Other helpful methods */
  public void setLocked(boolean l) { this.locked = l; }


}
```

Listing 3.54: This is our final-working ChestedDoor object.

The KnockDoor

Our intent here is to put together a counter and a door. An idea could be count the knocks that the player makes to the door: it could be a nice surprise if after a certain number of knocks the door pops out some coins (and it is a good idea for an hidden feature that the player has to discover by himself). So let's create the related trait:

```
/* Puts together a door and an enchantment */
public interface TKnockDoor extends TDoor, TCounter {

  /**
    Every knock makes the counter increment.
    If the limit is reached, more coins are released.
  **/
  default int knock() {
    int coins = TDoor.super.knock();
    incrementCounter();
    if(hasReachedLimit()) {
      System.out.print("Ohh! A special drop for you! ");
      coins += releaseCoins();
    } else {
      //Let's give a suggestion to the player
      System.out.print("Don't challenge me... ");
      int c = getLimit();
      String sug = "never knock a door more then "+c+" times.";
      System.out.println(sug);
    }

    return coins;
  }
}
```

Listing 3.55: The TKnockDoor trait.

Again, we create a class that implements the trait:

```
public class KnockDoor implements TKnockDoor {

  /* Fields for the door */
  private boolean locked;

  /* Fields for the counter */
  private int counter;
  private int limit;

  /* Glue Code for TDoor */
  public boolean getLocked() { return this.locked; }

  /* Glue code for TCounter */
  public int getCounter() { return this.counter; }

  public void setCounter(int c) { this.counter = c; }

  public int getLimit() { return this.limit; }

  /* Glue code for coin management */
  public int getDoorMaxCoins() { return 120; }

  public int getCounterMaxCoins() { return 500; }


  /* Constructor */
  public KnockDoor (boolean l, int li){
    setCounter(0);
    setLocked(l);
    setLimit(li);
  }

  /* Other helpful methods */
  private void setLocked(boolean l) { this.locked = l; }

  private void setLimit(int l) { this.limit = l; }

}
```

Listing 3.56: This is our final-working KnockDoor object.

3.5.5 Creating the DoorsRoom

Now we have three different kind of doors perfectly working thanks to our middle-traits (TEnchantedDoor, TChestedDoor and TKnockDoor). We have to create some kind of "room" object that can finally instantiate the doors we created before.

We could think of a TRoom trait that provides general methods for a room and then a class DoorsRoom that implements it and adds other features to the trait (the doors for sure!).

This is a good approach, but in order to not introduce too much code (that could be confusing all in once) we will just implement directly the DoorsRoom class (notice the use of TDoor as a type):

```java
public class DoorsRoom { //implements TRoom

  private TDoor leftDoor;
  private TDoor rightDoor;
  private TDoor frontDoor;

  /* Constructor */
  public DoorsRoom(TDoor l, TDoor r, TDoor f) {
    leftDoor = l;
    rightDoor = r;
    frontDoor = f;
  }

  /* Getters */
  public TDoor getLeftDoor() { return leftDoor; }
  public TDoor getRightDoor() { return rightDoor; }
  public TDoor getFrontDoor() { return frontDoor; }

}
```

Listing 3.57: A DoorsRoom has three doors inside of it.

3.5.6 The Game and the Player

Now that we have a room where our doors can work properly, we can finally put everything together by creating two simple classes:

- A Game class that provides some general variables and has a pointer to the rooms of the game.
- A Player class that is pointed by the Game class. Again, in this case, a TPlayer trait and then some classes like PremiumPlayer, FemalePlayer and MalePlayer, DemoPlayer etc. are a perfect application of our trait-approach to promote the code reuse. For simplicity we will just provide a Player class.

Here's our (very simple) Player class:

```java
public class Player { //implements TPlayer

  private final String nickname;
  private int bag = 150; //contains the coins

  /* Constructor */
  public Player(String n) {
    nickname = n;
  }

  /* Setters and getters */
  public int getCoins() { return bag; }
  public String getNickname() { return this.nickname; }


  /* Helpful methods */
  public void addInBag(int amount) {
    this.bag += amount;
  }

  public void removeFromBag(int amount) {
    this.bag -= amount;
  }

  public String toString() {
    String s = "I'm "+getNickname();
    s += " and i've got "+getCoins()+" coins in my bag.";
    return s;
  }

}
```

Listing 3.58: The Player keeps his coins into a bag.

Then the last class, Game:

```
public class Game {

  private Player player;
  private DoorsRoom doorsRoom; //should be an array
  private final String version = "0.0";

  /* Constructor */
  public Game(Player p, DoorsRoom dr) {
    player = p;
    doorsRoom = dr;
  }

  /* Setters and getters */
  public Player getPlayer() { return player; }
  public DoorsRoom getDoorsRoom() { return doorsRoom; }

}
```

Listing 3.59: The Game class links the player and the game space.

3.5.7 Let's give it a try

We are ready to test if our code works well or not. Here is the main we are going to launch. It is quite self explanatory.

```
public class GameTest{
  /** Just a brief alias for System.out.println */
  public static void pr(Object s) {
    System.out.println(s);
  }

  public static void main(String[] args) {
    //First some initialization
    int coins = 0; //Will be used later

    Player bob = new Player("Bobby");
```

```java

    //Left door = knock Door (locked, count = 2)
    //Right door = chested door (unlocked)
    //Front door = enchanted door (unclocked)
    DoorsRoom dr = new DoorsRoom(
        new KnockDoor(true, 2),
        new ChestedDoor(false),
        new EnchantedDoor(false));

    Game game = new Game(bob, dr);

    pr("@-@-@-@ THE GAME STARTS @-@-@-@");
    pr("You are in a room. It has three doors in it!");


    pr("\n--------------------------------------------------");
    pr(bob);
    pr("--------------------------------------------------");

    // TEST THE LEFT DOOR
    for(int i = 0; i <= 1; i++) {
      pr("\n["+bob.getNickname()+" knocks the left door]");
      coins = game.getDoorsRoom().getLeftDoor().knock();
      bob.addInBag(coins); //coins to the player
    }

    pr("\n["+bob.getNickname()+" tries to open the left door]");
    coins = game.getDoorsRoom().getLeftDoor().open();
    if(coins > 0) bob.addInBag(coins); //coins to the player

    pr("\n--------------------------------------------------");
    pr(bob);
    pr("--------------------------------------------------");

    // TEST THE RIGHT DOOR
    pr("\n["+bob.getNickname()+" tries to open the right door]");
    coins = game.getDoorsRoom().getRightDoor().open();
    if(coins > 0) bob.addInBag(coins); //coins to the player

    pr("\n--------------------------------------------------");
    pr(bob);
    pr("--------------------------------------------------");

    // TEST THE FRONT DOOR
    pr("\n["+bob.getNickname()+" tries to open the front door]");
    coins = game.getDoorsRoom().getFrontDoor().open();
    if(coins > 0) bob.addInBag(coins); //coins to the player

    pr("\n-------------------------------------------------------");
    pr(bob);
    pr("-------------------------------------------------------");
  }


}
```

Listing 3.60: This test will let us see if everything works properly.

Here's the output from the `main` we just saw:

```
@-@-@-@ THE GAME STARTS @-@-@-@
You are in a room. It has three doors in it!

-------------------------------------------------------
I'm Bobby and i've got 150 coins in my bag.
-------------------------------------------------------


[Bobby knocks the left door]
Door says: How you dare, I am the one who knocks!
Don't challenge me... never knock a door more then 2 times.

[Bobby knocks the left door]
Door says: How you dare, I am the one who knocks!
Ohh! A special drop for you! You got 364 coins.

[Bobby tries to open the left door]
This door is locked.

-------------------------------------------------------
I'm Bobby and i've got 514 coins in my bag.
-------------------------------------------------------


[Bobby tries to open the right door]
The door has been opened!
You got 49 coins.
The chest is now opened! You got 141 coins from the chest.

-------------------------------------------------------
I'm Bobby and i've got 704 coins in my bag.
-------------------------------------------------------


[Bobby tries to open the front door]
The door has been opened!
You got 48 coins.

This is an enchantment!
```

```
"If the luck is up, of coins you'll have a cup,
but if no luck you got, you are gonna lose a lot."
You lost 123 coins!

______________________________________________________

I'm Bobby and i've got 704 coins in my bag.
______________________________________________________
```

Listing 3.61: The result of our main.

Everything went as expected. All the methods that we have called were correctly executed without any misdirection.

3.5.8 Zoom-out: the class diagram

Perfect! Everything works great. Let's make a zoom-out and see what we have done thanks to this class diagram (Figure 3.2).

As we can see, middle-traits make possibile to compose different features just in the way we want to.
Every trait is re-usable inside the project and every feature is stand-alone so we could for example use a TCounter in a totally different situation inside the game.
The composition of the classes is quite clear and isn't redudant at all.
The only negative consequence is a lot of glue code inside the implementing classes, and, sometimes, it is even repeated. We have anyway to consider that most of IDE can easly auto-generate setters and getters (that are basically all the glue code we used). It is anyway a good thing the fact that every class defines how to access to his own a variables. For instance, in this very example, the getDoorMaxCoins() assumes different return-values based on the kind of the door.

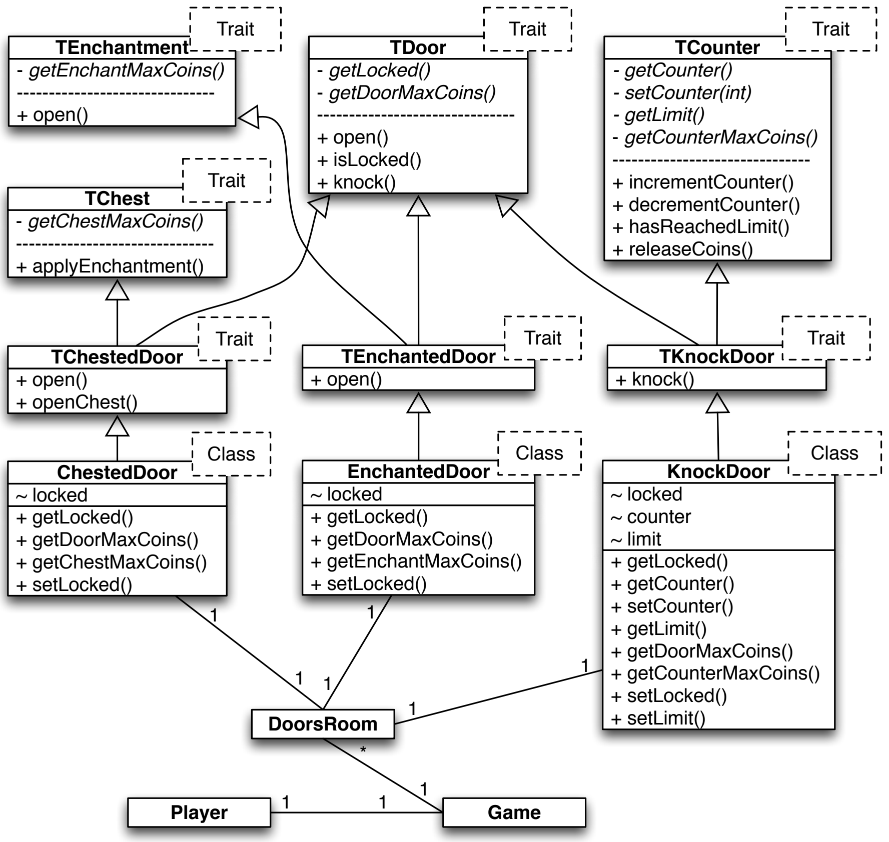

Figure 3.2: The class diagram of our game-example.

3.6 Conclusions

Traits (and mixins) are an alternative to inheritance and have the goal to help the developer in thinking to the elements of a project as stand-alone units of reuse that can be changed and remodelled when putted together. We developed some programming patterns with the aim to promote trait-oriented programming in Java 8. As it is usual with traits, it seems that our approach has an impact on code modularity, which implies more reusability and maintainability. We also hilighted some drawbacks of the approach, mainly due the features of the language (however, it might be possible to improve the patterns we presented).

3.6.1 Comparison with other solutions

The idea that Java 8 may promote a trait/mixin-oriented programming is not only ours, even though its development is at early stages. Here we compare our approach with other two (the most complete ones we found so far on-line).

From Kerflyn's Blog

The first solution we are going to see is from Kerflyn's Blog [Blo12].
The author is intrested in bringing mixins inside Java and so his work is stricly related to the state of the objects. Firstly his idea was to define an interface that deals with a boolean field thanks to two methods: the intresting part is the fact that those methods are from a static final class. This solution (as clearly pointed out from the author) is not thread-safe and is subject to memory leak. Here's the code:

```java
public interface SwitchableMixin {

  default boolean isActivated() {
    return Switchables.isActivated(this);
  }

  default void setActivated(boolean activated) {
    Switchables.setActivated(this, activated);
    }

}
```

Listing 3.62: This interface deals with a field.

And here is a class that provides a custom isActivated() and setActivated() methods:

```java
public final class Switchables {

  private static final Map<SwitchableMixin,
        SwitchableDeviceState> SWITCH_STATES = new HashMap<>();

  public static boolean isActivated(SwitchableMixin device) {
    SwitchableDeviceState state = SWITCH_STATES.get(device);
    return state != null && state.activated;
  }

  public static void setActivated(SwitchableMixin d, boolean a){
    SwitchableDeviceState state = SWITCH_STATES.get(d);
      if (state == null) {
        state = new SwitchableDeviceState();
        SWITCH_STATES.put(d, state);
    }
    state.activated = a;
  }

  private static class SwitchableDeviceState {
    private boolean activated;
  }
}
```

Listing 3.63: A class that connects mixins and their states.

The SwitchableDeviceState is the field container and is a static nested class. Also, the Switchables class provides a structure that actually composes a SwitchableMixin with his SwitchableDeviceState. Here is an use of the above implementation:

```java
private static class Device {}

private static class DeviceA extends Device
        implements SwitchableMixin {}

private static class DeviceB extends Device
        implements SwitchableMixin {}

public class TestDevice {
  public static void main(String[] args) {
    DeviceA a = new DeviceA();
    DeviceB b = new DeviceB();

    a.setActivated(true);

    assertThat(a.isActivated()).isTrue();
    assertThat(b.isActivated()).isFalse();
  }
}
```

Listing 3.64: An use case for the above implementation.

As the author says, this implementation seems to work well. But Brian Goetz suggested an use of this implementation that doesn't.

```java
interface FakeBrokenMixin {
  static Map<FakeBrokenMixin, String> backingMap
    = Collections.synchronizedMap(
      new WeakHashMap<FakeBrokenMixin, String>());

  default String getName() {
    return backingMap.get(this);
  }

  default void setName(String name) {
    backingMap.put(this, name);
  }
}

interface X extends Runnable, FakeBrokenMixin {}

public class Test {

  public static X makeX() {
    return () -> System.out.println("Hello! I'm a running X.");
  }

  public static void main(String[] args) {
    X x1 = makeX();
    X x2 = makeX();
    x1.setName("x1");
    x2.setName("x2");

    System.out.println(x1.getName());
    System.out.println(x2.getName());
  }
}
```

Listing 3.65: A not working solution.

The FakeBrokenMixin does both what Switchables and SwitchableMixin did before. We don't have two separated classes because the association is between a String and a FakeBrokenMixin, so we don't need another inner interface to

define a new type (that was SwitchableDeviceState before).
Given that this code fits with the author's solution, we have an interface X that is a functional interface (because it extends Runnable that is a functional interface itself). Note that as we outlined in Section 2.2.1 even if X has two defaults methods (getName() and setName()) that are inherited from FakeBrokenMixin interface, it is still a functional interface because the only method that lacks a body (is abstract) is run() from Runnable interface.

The Test class has two methods. makeX() is a static method that returns an instance of X. Because X is a functional interface, we can instantiate it thanks to a lambda expression (line 20).
To make it clear: makeX() method returns an X object that is both a Runnable and a FakeBrokenMixin. The run() method required from the Runnable interface is defined thanks to the lambda expression at line 20 (that also instantiates the X object). This means that a variable var of type X can be used in a call like var.run() and would print "*Hello! I'm a running X.*". The main just creates two instances of X, gives a different name to each of them and then prints those names. The output we would expect is:

```
x1
x2
```

Listing 3.66: Expected output from Test execution.

Instead, we get:

```
x2
x2
```

Listing 3.67: The real output from Test execution.

Why this happens? It looks like x1 and x2 are the same object. In fact, if we try to print the pointer of these variables, it is the same (something like Test$$Lambda$1@72ea2f77).

Each call to makeX() seems to give a singleton from the same anonymous inner class.
We report the explanation given by the author of the post [Blo12].

The complete translation of the lambda expression is not really done at compile time. In fact, this translation is done at compile time and runtime. The compiler javac put in place of lambda expression an instruction recently added to JVM : the instruction invokedynamic (JSR292). This instruction comes with all the necessary meta-information to the translation of the lambda expression at runtime. This includes the name of the method to call, its input and output types, and also a method called *bootstrap*. The bootstrap method aims to define the instance that receive the method call, once the JVM execute the instruction invokedynamic. In presence of lambda expression the JVM uses a particular bootstrap called *lambda metafactory method*.

In our example, the body of the lambda expression is converted into a private static method. Thus, () -> System.out.println("Hello! I'm a running X."); is converted in Test into:

```
private static void lambda$0() {
    System.out.println("Hello! I'm a running X.");
}
```

Listing 3.68: How the lambda expression return is converted by the JVM.

When you run the program, once the JVM tries to interpret the invokedynamic instruction for the first time, the JVM call the lambda metafactory method described above. In our example, during the first call to makeX(), the lambda metafactory method generates an instance of X and links the method run() dynamically to the method lambda$0(). The instance of X is then stored in the memory. During the second call to makeX(), the instance is brought back. There you get the same instance as during the first call.

Note that this problem isn't really related to our trait-approach but it is something to keep in mind while using lambda expressions.

As there is no fix or workaround for this problem, the author provided an alternative way to deal with fields. Is the *virtual field pattern* that we used all along in this work: we request the access to a field by defining a required method that will be implemented in the class that uses the trait.

```java
interface Trait {
  public Object getField();

  default void m() {
    System.out.println(getField());
  }
}

public class Test implements Trait {

  private String str;

  public getField() {
    return this.str;
  }
}
```

Listing 3.69: The virtual field pattern.

Thanks to the Kerflyn's work we have seen how have a complete stateful approach is difficult: mind that the virtual field pattern isn't really a mixin-related solution as it is the common way to deal with fields in the original proposed trait model.

From Reidar Sollid's Blog

Another solution from the community is taken by Reidar Sollid's Blog [Sol13]. His intent was to define an IntQueue that collects integers in a queue. Then the author wanted to apply different features to the queue (like dubling the elements while putting them into the queue). He defined the queue in this way:

```java
public interface IntQueue {
    Queue getQueue();

    default int get() {
        return getQueue().remove();
    }

    default void put(int x) {
        getQueue().add(x);
    }

}
```

Listing 3.70: The IntQueue interface.

This is exactly the way we defined our traits. We can see the use of the virtual field pattern to access the queue and we can see the implementation of two default methods to work with the queue. Then, he defined two other interfaces (traits) that can provide different features to the queue:

```java
public interface WithDoubling extends IntQueue {
    default void put(int x) {
        IntQueue.super.put(x * 2);
    }
}
```

Listing 3.71: Every element putted in the queue is doubled.

```java
public interface WithFiltering extends IntQueue {
    default void put(int x) {
        if (x > 0) IntQueue.super.put(x);
    }
}
```

Listing 3.72: Filters the elements to be putted in the queue.

We will create a class BasicIntQueue that provides the queue and the glue methods for the IntQueue interface. As we already know if we create a class that implements IntQueue, WithFiltering and WithDoubling together we will have a conflict on the put() method.

The author found a solution in the decorator pattern [GHJV94]: Filtering, Doubling, etc. will now become decorations for the BasicIntQueue class. This solution seems to work well but it doesn't use any Java 8 feature at all! The main point is that the authors has in mind Scala [gro], where traits are in fact mixing and he try to reconstruct Scala mixins in Java 8. Actually the decorator is easy implementable with mixing, but not with traits, as trait composition is a true composition, instead mixin composition is immediately transformed in an inheritance hierarchy. Using our model, we could create a TDouble and a TFilter traits that work in a stand-alone way. Then we can create a TFilterDoubleQueue that uses the features provided from IntQueue, TFilter an TDouble for defining a new put() method. Notice that this approach gives the opportunity to extend the queue in a second moment (for example by adding a TIncrement trait). Not only, it's possibile to redefine some of the features already binded with the queue (for instance, we can override the method provided from the TFilter trait and change the filter).

3.6.2 Combining the reuse

In our application we always used a only-trait style as it was our purpose to introduce this new approach. But Java always offers the single inheritance and, it could be possibile, sometimes, use in a combined way single inheritance and our trait-approach to bring at his maximum the reusability. Let's take our example in Section 3.5. Looking at Figure 3.2 we notice the repetition of the glue code in the below classes: maybe it could have been re-grouped in a class (together with the related fields) called CommonFields. It is up to the developer to understand when single inheritance can be helpful and when it is only a not convenient limitation to the reuse.

3.7 What's next

While developing the trait-approach we didn't consider the aspect of the security. The first thing that has to be done is to put our code under pressure with built examples and robustness tests. Not only, more study cases have to be developed in order to find any other possibile difficult situation for the model.

Our intent is to go from a trait-oriented programming style in Java 8 to a proper extension of Java with traits as part of the language, with a semantics based on our programming style.
In order to go in this direction, it is necessary to explore in depth other proposals of Java with traits [BDG08, SD05] and languages such as Scala [gro, PvdB12]; moreover, we also need to explore other ways to deal with fields different from the *virtual field pattern*, which is our approach at the moment. A fundamental point in this study will be the design of an appropriate type system.

Bibliography

[BDG08] Viviana Bono, Ferruccio Damiani, and Elena Giachino. On traits and types in a java-like setting. In Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 367–382. Springer, 2008. `http://dblp.uni-trier.de/db/conf/ifipTCS/ifipTCS2008.html#BonoDG08`.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. `http://dx.doi.org/10.1016/j.cl.2007.05.003`.

[Blo12] Kerflyn's Blog. Java 8: Now you have mixins? `http://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/`, June 2012.

[DNS$^+$06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2:331–388, 2006.

[FF98] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. *SIGPLAN Not.*, 34(1):94–104, September 1998. `http://doi.acm.org/10.1145/291251.289432`.

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, UK, 1999. Springer-Verlag. `http://dl.acm.org/citation.cfm?id=645580.658808`.

[GF12] B. Goetz and R. Field. Featherweight Defenders: A formal model for virtual extension methods in Java. `http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf`, March 2012.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[Goe11] B. Goetz. Interface evolution via virtual extensions methods. `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf`, June 2011.

[gro] The Scala group. Scala website. `http://www.scala-lang.org/`.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[Ora] Oracle. *The Java Tutorials*. `http://docs.oracle.com/javase/tutorial/java/javaOO/`.

[PvdB12] Urs Peter and Sander van den Berg. Java 8 vs Scala: a feature comparision. `http://www.infoq.com/articles/java-8-vs-scala`, June 2012.

[SD05] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP '05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.

[SDNB03] N. Schärli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.

[Sol13] Reidar Sollid. Java 8 and mixin with default methods. `http://reidarsollid.com/2013/03/28/java-8-and-mixin-with-default-methods/`, March 2013.