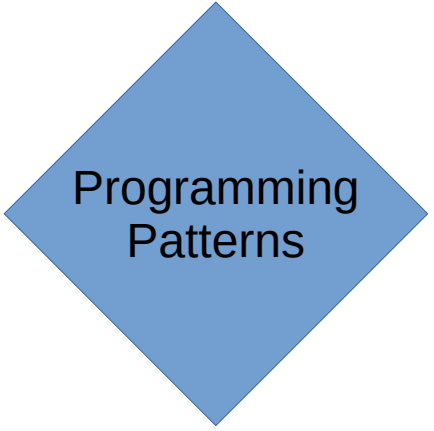
SWEN423 - Lecture 13



Class Less Java

Can you write in Java without ever using the keyword

'Class'

Let's begin

```
public interface Main{
  public static void main(String[]a){
    System.out.println("Hello world without 'class'");
    }
  }
```

Let's begin with the usual Point

```
class Point{int x; int y;//(1) the worst
  public int sumCoords(){return x+y;}
  public int x(){return x;} public int y(){return y;}
  public Point(int x,int y){this.x=x;this.y=y;}
  //equals, hashCode and toString are the Object one.
  }

interface Point{int x();int y();//(2) better extensibility
  default int sumCoords(){return x()+y();}
  static Point of(int x,int y){
    return new Point(){public int x(){return x;} public int y(){return y;}};
    //equals, hashCode and toString are the Object one.
    }}

record Point(int x,int y){//(3) better usability + syntax, not extensible
  public int sumCoords(){return x+y;}
  }//equals, hashCode and toString make sense

//When there is no need of extensibility, use records!
```

Extending Point with Classes

```
class Point{int x; int y;
  public int sumCoords(){return x+y;}
  public int x(){return x;} public int y(){return y;}
  public Point(int x,int y){this.x=x;this.y=y;}
  }

class Point3D extends Point{int z; //add a field
  public int sumCoords(){return super.sumCoords()+z;} //call super
  public int z(){return z;}
  public Point3D(int x,int y,int z){super(x,y);this.z=z;} //call super constructor
}

class ColPoint extends Point{String color; //add a field
  public String color(){return color;}
  public ColPoint(int x,int y,String color){super(x,y);this.color=color;}//super constr
  }

class ColPoint3D extends Point3D,ColPoint3D{}//WRONG!
```

Extending Point with Interfaces

```
interface Point{int x();int y();
  default int sumCoords(){return x()+y();}
  static Point of(int x,int y){
    return new Point(){public int x(){return x;} public int y(){return y;}};}}

interface Point3D extends Point{int z();//add a getter
  default int sumCoords(){return Point.super.sumCoords()+z();} //call super
  static Point3D of(int x,int y,int z){
    return new Point3D(){//no super constructor, manual handling of all fields
      public int x(){return x;}public int y(){return y;}public int z(){return z;}};}}

interface ColPoint extends Point{String color();
  static ColPoint of(int x,int y,String color){
  return new ColPoint(){
    public int x(){return x;}public int y(){return y;}public String color(){return color;}};}}

interface ColPoint3D extends Point3D,ColPoint{ //MULTIPLE INHERITANCE
  default int sumCoords(){return Point3D.super.sumCoords();} //call super
  static ColPoint3D of(int x,int y,int z,String color){
    return new ColPoint3D(){//manual handling of all fields is good for multiple inheritance
      public int x(){return x;} public int y(){return y;}
      public int z(){return z;} public String color(){return color;}};}}
```

Extending Point with Interfaces

```
//Interface + Records for the best result!
interface Point{int x();int y();
  record Of(int x,int y)implements Point{}
  default int sumCoords(){return x()+y();}
  //PointRI.Of will have good equals, hashCode and toString
  }

interface Point3D extends Point{int z();
  record Of(int x,int y,int z)implements Point3D{}
  default int sumCoords(){return Point.super.sumCoords()+z();}
  }

interface ColPoint extends Point{String color();
  record Of(int x,int y,String color)implements ColPoint{}
  }

interface ColPoint3D extends Point3D,ColPoint{
  record Of(int x,int y,int z,String color)implements Point3D,ColPoint{}
  default int sumCoords(){return Point3D.super.sumCoords();}
  }
//It works because records define getters named x(), y() and so on..
//and they have the same names of the getters of the interfaces
//It would not work if we also need setters :-(
```

```
//usage
new ColPoint3D.Of(0, 2, 3, "blue");
```

Using interfaces to split large code

- If you have a class with many methods, and some of those are complex and have a lot of private sub-methods, you can use interfaces to split you code in more manageable units

```
interface Complex{int field1(); int field2();
  record Of(int field1,int field2) implements M1,M2,M3,M4{}
  int m1(int a);
  int m2(int b);
  int m3(int c);
  int m4(int d);
  }
//in file M1
interface M1 extends Complex{
  default int m1(int a){return this.foo()+this.bar(a);}//example of complex code
  private int foo(){return 0;}//example of private auxiliary methods
  private int bar(int tmp){return tmp;}
  }
//in file M2
interface M2 extends Complex{
  default int m2(int b){..}
  private int foo(){..}
  ..}
//in file M3
interface M3 extends Complex{
  default int m3(int c){..}
  ..}
//in file M4
interface M4 extends Complex{
  default int m4(int d){..}
  ..}
```

And now... just as an exercise

- Can we do more?
- Can we code without the keywords
  - class, record and new?
- It is easy if the class have a single field!

```
interface Person{
  String name();
  default int exampleMeth(){return name().length();}
  static Person of(String name){return ()->name;}
}
```

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
   interface Map<A,B>{Pair<A,B> of(A a,B b);}
   interface PairA<A,B> extends Pair<A,B>{A a();}
   interface PairB<A,B> extends Pair<A,B>{B b();}
   interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
   static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
   static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
   static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
   default A a(){return map((a,b)->pairA(()->a)).a();}
   default B b(){return map((a,b)->pairB(()->b)).b();}
   default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
   static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
   }
```

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
   interface Map<A,B>{Pair<A,B> of(A a,B b);}
   interface PairA<A,B> extends Pair<A,B>{A a();}
   interface PairB<A,B> extends Pair<A,B>{B b();}
   interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
   static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
   static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
   static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
   default A a(){return map((a,b)->pairA(()->a)).a();}
   default B b(){return map((a,b)->pairB(()->b)).b();}
   default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
   static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
   }
```

Pair have 3 methods, all with default impl.

Thus, Pair can not be lambda-instantiated

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
   interface Map<A,B>{Pair<A,B> of(A a,B b);}
   interface PairA<A,B> extends Pair<A,B>{A a();}
   interface PairB<A,B> extends Pair<A,B>{B b();}
   interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
   static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
   static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
   static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
   default A a(){return map((a,b)->pairA(()->a)).a();}
   default B b(){return map((a,b)->pairB(()->b)).b();}
   default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
   static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
   }
```

PairA, PairB and PairM all override a method by making it abstract. They can now be lambda-instantiated

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
  interface Map<A,B>{Pair<A,B> of(A a,B b);}
  interface PairA<A,B> extends Pair<A,B>{A a();}
  interface PairB<A,B> extends Pair<A,B>{B b();}
  interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
  static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
  static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
  static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
  default A a(){return map((a,b)->pairA(()->a)).a();}
  default B b(){return map((a,b)->pairB(()->b)).b();}
  default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
  static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
  }
```

usually, the type of the lambda can be inferred by the context. Here it would not work: Our lambda candidates all extends Pair<A,B>

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
  interface Map<A,B>{Pair<A,B> of(A a,B b);}//given a,b  makes a Pair<A,B>: a factory!
  interface PairA<A,B> extends Pair<A,B>{A a();}
  interface PairB<A,B> extends Pair<A,B>{B b();}
  interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
  static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
  static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
  static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
  default A a(){return map((a,b)->pairA(()->a)).a();}
  default B b(){return map((a,b)->pairB(()->b)).b();}
  default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}//not for “real” Pairs

  //Convenience factory method          All our “real” Pairs, are instances of PairM!
  static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}//closure on a,b!
  }//This code does not make a Map<A,B>, but a PairM<A,B>:
 //a function of type ((A,B)->Pair<A,B>)->Pair<A,B>
 //“if you give me a function from A,B to Pair<A,B>, I produce a Pair<A,B>, by passing a,b”
```

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
   interface Map<A,B>{Pair<A,B> of(A a,B b);}
   interface PairA<A,B> extends Pair<A,B>{A a();}
   interface PairB<A,B> extends Pair<A,B>{B b();}
   interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
   static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
   static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
   static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
   default A a(){return map((a,b)->pairA(()->a)).a();}//make a Pair that is only able
   default B b(){return map((a,b)->pairB(()->b)).b();}//to give the 'a' component
   default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}
  //pairA(()->a)): a Pair where we override 'a()' to return the value 'a'
  //Convenience factory method
   static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
   }
```

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
  interface Map<A,B>{Pair<A,B> of(A a,B b);}
  interface PairA<A,B> extends Pair<A,B>{A a();}
  interface PairB<A,B> extends Pair<A,B>{B b();}
  interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
  static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
  static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
  static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
  default A a(){return map((a,b)->pairA(()->a)).a();}
  default B b(){return map((a,b)->pairB(()->b)).b();}//same for b
  default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
  static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
  }
```

Web tool: The last question

```
interface Pair<A,B>{
  //Auxiliary interfaces
   interface Map<A,B>{Pair<A,B> of(A a,B b);}
   interface PairA<A,B> extends Pair<A,B>{A a();}
   interface PairB<A,B> extends Pair<A,B>{B b();}
   interface PairM<A,B> extends Pair<A,B>{Pair<A,B> map(Map<A,B> m);}

  //Convenience methods to create instances of those interfaces
   static <A,B> PairA<A,B>pairA(PairA<A,B> id){return id;}
   static <A,B> PairB<A,B>pairB(PairB<A,B> id){return id;}
   static <A,B> PairM<A,B>pairM(PairM<A,B> id){return id;}

  //The core of the solution
   default A a(){return map((a,b)->pairA(()->a)).a();}
   default B b(){return map((a,b)->pairB(()->b)).b();}
   default Pair<A,B> map(Map<A,B> m){return m.of(a(),b());}

  //Convenience factory method
   static <A,B> Pair<A,B> of(A a,B b){return pairM(m->m.of(a, b));}
   }
```

And now we are sorted!

- In the same way we can implement Pair<A,B>, we can implement Tuple3<A,B,C>, Tuple4...
- With Tuples, we can represent any class as an interface with a single field of type TupleN<A,...>

```
interface Person{
  static Person of(String name,Integer age, Point state){
    return ()->Tuple3.of(name,age,state);
    }
  Tuple3<String,Integer,Point> state();
  default String name(){return state()._1();}
  default Integer age(){return state()._2();}
  default Point position(){return state()._3();}
  default String repr(){return "";}//more methods here
  }
```

- Both 'new' and 'class' are redundant keywords :-)

Do not code like this!

- But, from a formal perspective, 'new' and 'class' could be not modeled
- We could make a more minimal FJ:

  e ::= x | (xs)->e | e.m(es)
- And this is basically lambda calculus with a class table. In some sense, we reached a formal unification of Object oriented languages and functional languages