Part 1

The goal is to write classes/traits which element allow to write a hierarchy of consumers & consumed. We want to avoid the possibility that an animal that is "designed" to eat plants accepts to eat meat and another way around. Effectively there are two hierarchies of top classes/traits. One with the Food class/ trait at the top and another with the Animal at the top (they actually mix a bit for specific classes).

The implementation of the method eat should only exists in the class/trait Animal.

Following entities exist: Food, Carrot, Rabbit, Wolf, Meat, Plants, Animal.

```
val c: Plants = new Carrot

val r = new Rabbit
val w = new Wolf
val a: Animal = r
val f: Food = r
// this two should compile and run w/o problems
r.eat(c)
w.eat(r)
// whereas these should not compile
r.eat(r)
w.eat(c)

Expected result:
  Rabbit crunches the  Carrot
  Wolf tears apart the Rabbit
```

Alternative Part 1

★The goal would be to write “know where failed” KWF monad
The wrapping class would contain:
value of type double, and:

★*single counter from the moment when the computation failed*

Either of the two could work actually

★*two counters, one counting number of successful operations that succeeded from the start, another number of operations that were skipped because of failure. (Obviously second counter == 0 would signify success.)*

★***The magic - clarity and easiness of the solution - will happen when you realise that the counter (or counters) - and it’s operation needs to form monoid***

★The companion object should be equipped with higher order, taking two `double => KWF` and composes them like this: ***
`compose(fa, fb)` where fa and fb are two functions from Double to KWF

***** you can add receiver conversion to define handy composition syntax: fa >>> fb >>> fc >>> fd**

Part 2

It is about expressions matching. We have a data:

```
 val data = List( Map("name" -> "Jan", "fname" -> "Kowalski", "age" -> "45"),
    Map("company" -> "ABB", "origin" -> "Sweden"),
    Map("name" -> "Katarzyna", "fname" -> "Nowak", "age" -> "45"),
    Map("company" -> "F4", "origin" -> "Poland"),
    List("Cos", "innego"),
    Map("company" -> "Salina Bochnia", "origin" -> "Poland"),
    Map("company" -> "AGH", "origin" -> "Poland"),
    Map("name" -> "Krzysztof", "fname" -> "Krol", "age" -> "14")
  )
```

And we want to have function that will extract from it all companies.
During the extraction, a class `Company` (trivial case class can be used) should be created.
`getCompanies(data)` should result in:

```
List(Company(ABB,Sweden), Company(F4,Poland), Company(Salina Bochnia,Poland),
Company(AGH,Poland))
```