Exercise sheet 11— Solutions

1 Modelling a wallet (old exam question)

```
final case class Wallet(
  amountMoney: Int,
  numberOfDocuments: Int,
)
// --- later ---
def transformWallet(w: Wallet): Wallet =
  if w == null then
    changeWallet(w)
  else
    throw new RuntimeException("no wallet");
```

Possible improvements:

- `amountMoney` and `numberOfDocuments` can easily be mixed up, because they have the same type. A solution could be to introduce a currency type (which also helps with multiple currencies and similar…).
- We never use `null`. If our input should be able to be possibly empty, use `Option`.
- "Boolean blindness" – Inside the `if` branches, it isn't obvious anymore, which one was the positive and which one the negative branch. Using explicit pattern matching (e.g on Option) or higher order functions of the used type (e.g. `map`) makes this more clear (the code actually has a bug: changeWallet is called, if the wallet *is* `null`)
- We never use exceptions. If it is possible, that the operation fails, use `Either` for fail-fast behaviour or `Validated` for accumulation of errors.

One possible improved version:

```
case class Money(amount: Int, currency: String)
final case class Wallet(
  amountMoney: Money,
  numberOfDocuments: Int,
)
// --- later ---
def transformWallet(w: Option[Wallet]): Either[String, Wallet] =
  w.map(changeWallet).toRight("no wallet")
```

2 Modelling a customer (old exam question)

2.1 Type "Customer"

```
final case class NonEmptyList[A](head: A, tail: List[A])

enum Customer:
  case Private(name: String, phoneNumber: Option[String])
  case Business(name: String, phoneNumber: NonEmptyList[String])
```

2.2 Possible problem

Name and phone number are of the same type, even though they describe fundamentally different concepts. It would be better, to use a separate type for the phone number, which can also check the validity.

3 Recap: Parametricity (old exam question)

```
def p2[A,B,C,D](a: A, b: B)(f: (A,B) => C, g: (A,C) => D): D =
  g(a, f(a,b))
```

The explanation is also the solution for exercise part b):

As the types are variable, no operations beside the given ones can be executed. There is no possibility to instantiate the variable types `C` and `D`. As a `D` has to be returned, it has to be created using the given functions.

4 Recap: Lazy Evaluation (old exam exercise)

4.1 Strict parameters vs. call-by-name vs. lazy val

Strict parameters are always evaluated exactly once *before* executing the method body.

Call-by-Name parameters are only evaluated, if the evaluation of the method body reaches a usage point of them. The result is then not cached, so that a call-by-name parameter may be evaluated several times.

lazy val is only evaluated when the value is actually used, like call-by-name, but the evaluation result is stored, so that it is calculated at most once.

4.2 Early-Stopping for foldRight

We are given the following function signature:

```
def foldRight[B](z: B)(f: (A, B) => B): B = this match
  case Cons(x, xs) => f(x, xs.foldRight(z)(f))
  case Nil => z
```

To change it to allow for early stopping, we only have to modify the signature, making some parameters call-by-name:

```
def foldRightLazy[B](z: => B)(f: (A, => B) => B):B = this match
  \\ ... same ...
```

As the recursive call is in the second parameter to `f`, it can be prevented by making this parameter call-by-name. Now the call only happens, if the given function uses it. This way the function itself can decide, if the recursion should continue. The `z` is not required to be call-by-name to allow for this behaviour, but this way it is also only calculated, if the fold iterates through the whole list.

A solution without using laziness is also possible, but it requires more changes. Instead of deciding in the given function, if we should continue calculation, we pass a separate function for that. This function takes the current element and returns an Option of the result type. If it is empty, we continue with the calculation. But if it contains a value, we return that without recursing further.

```scala
def foldRightLazy[B](z: => B)(p: A => Option[B])(f: (A, B) => B): B = this match
  case Cons(x, xs) => p(x) match
    case None  => f(x, xs.foldRight(z)(p)(f))
    case Some(res) => res
  case Nil => z
```

4.3 foldRight — Calls

These are calls matching the two solutions presented above, which multiply the elements in the list and on encountering a 0 abort and return 0 directly:

```scala
intList.foldRightLazy(1)((a, b) => if a == 0 then 0 else a * b
```

```scala
intList.foldRightLazy(1)(if i == 0 then Some(0) else None)(_ * _)
```