Software Engineering: Tutorial 9

David Voigt

October 23th, 2022

Agenda

1. Common mistakes in the last homework
2. Subtyping
3. Variance
4. exercise

Common mistakes: Fixation on hash maps

```scala
class StdInStore(
    private var myStorage: HashMap[String, String]
) extends Store {
  def readInput(): Unit = {
    var key = ""
    while (key != ":q") {
      key = readLine("Enter Key (:q to stop input): ")
      if (key != ":q") {
        val value = readLine("Enter Value for previous key: ")
        myStorage = myStorage.updated(key, value);
      }
    }
  }
  def get(key: String): Option[String] = {
    myStorage.get(key)
  }
```

Common mistakes: Fixation on hash maps

```
class StdInStore(
    private var myStorage: HashMap[String, String]
) extends Store {
  def readInput(): Unit = {
    var key = ""
    while (key != ":q") {
      key = readLine("Enter Key (:q to stop input): ")
      if (key != ":q") {
        val value = readLine("Enter Value for previous key: ")
        myStorage = myStorage.updated(key, value);
      }
    }
  }
  def get(key: String): Option[String] = {
    myStorage.get(key)
  }
```

- The Store interface is not directly related to a hash map. Only the converse.

Common mistakes: Fixation on hash maps

```scala
class StdInStore extends Store {
def get(key: String): Option[String] =
    readLine(s"Enter a value for $key: ") match {
        case null | "" => None
        case s => Some(s)
    }
}
```

Common mistakes: Use regexes

- Instead of manually juggeling with indices by splitting strings, one may rather use regular expressions for parsing
- This really simplifies the parser for time and distance

Common mistakes: Use regexes

- Instead of manually juggeling with indices by splitting strings, one may rather use regular expressions for parsing
- This really simplifies the parser for time and distance

```scala
case class Distance(n: Int)
object Distance {
    private val pKilometer = raw"(\d+) km".r
    private val pMeter = raw"(\d+) m".r
    def parseDistance(s: String): Option[Distance] = s match {
        case pKilometer(n) => Some(Distance(n.toInt * 1000))
        case pMeter(n) => Some(Distance(n.toInt))
        case _ => None
    }
}
```

Subtyping & Variance

Wait... subtyping - never heard of it

Well yes, you have in Informatik 2 (maybe by another name):

```
class Item
class Buyable extends Item
class Book extends Buyable
```

- Here, Buyable is a **supertype** of Book, while Book is a **subtype** of Buyable (similarly for Item and Buyable)
- In Scala we may write this relation as Book <: Buyable <: Item
- How is this useful? A **subtype** may be used everywhere a **supertype** is expected!

```
def getPrice[T <: Buyable](b: T): Price
```

Variance: Definition

- Variance is directly related to subtyping and describes how a type constructor transform the ordering of types
- Let `T: * -> *` be type constructor, and `A` and `B` types
  - **Covariance**: `A <: B => T<A> <: T<B>`
  - **Contravariance**: `A <: B => T<B> <: T<A>`
  - **Invariance**: If neither covariant nor contravarient, then invarient

type- what now?

- Type constructors are similar to value constructors.
- Take for example the **value constructor** `Some`, `Some` takes a value argument and creates a value of type `Option[T]`, where `T` is the type of the value argument.
- Similarly, `Option` can be seen as a **type constructor** that takes one type as argument and returns a type. For example, the type `Int` applied to `Option` yields the type `Option[Int]`.
- `* -> *` is called a kind and is the “type” of type, that is, in this example, one type is expected and a new type is returned. `* -> *` is the the kind of the type constructor `Option`

Example: Mutable Array

- `Array` has the kind `* -> *`, that is, it takes one type parameter and returns a new type
- An mutable array has to be invariant. Why?
- Lets consider an an example array `Array[Animal]`, where `Cat <: Animal` and `Dog <: Animal`
  - **Contravariance**: `Array[Animal] <: Array[Cat]` and `Array[Animal] <: Array[Dog]`, but then a user reading from an `Array[Cat]` may encounter a dog!

    ```
    var a: Array[Cat] =...; val b: Array[Animal]=...; a = b // !!!
    ```
  - **Covariance**: `Array[Cat] <: Array[Animal]` and `Array[Dog] <: Array[Animal]`, but then a user might insert a Dog into an `Array[Cat]`!

    ```
    var a: Array[Animal] = ...; val b: Array[Cat]= ...
    a=b; a.insert(Dog()) // !!!
    ```
- Conclusion:
  - Read-only data types may be covariant
  - Write-only data types may be contravariant

Exercises

- https://github.com/se-tuebingen-exercises/tut7-exercise9
- `git clone git@github.com:se-tuebingen-exercises/tut7-exercise9.gi`