Warsztat 8 Programowanie w języku Scala

Kamil Celejewski 239642

Michał Suliborski

3. Abstract members

- Abstract members is a member of a class or trait if the member does not have a complete definition in the class. Abstract members are intended to be implemented in subclasses of the class in which they are declared.
- In Scala methods, values, variables, types, objects, traits, and classes are collectively called members.
- Member variables are known as instance variables in Java.
- *Abstract type* in Scala means a type declared (with the "type" keyword) to be a member of a class or trait, without specifying a definition. Classes themselves may be abstract, and traits are by definition abstract, but neither of these are what are referred to as *abstract types* in Scala. An abstract type in Scala is always a member of some class or trait, such as type T in trait Abstract.
- It is also possible to turn abstract type members into type parameters of classes and vice versa.

4. Existential types

- Existential types are a way of abstracting over types. They let you “acknowledge” that there is a type involved without specifying exactly what it is, usually because you don’t know what it is and you don’t need that knowledge in the current context.
- Existential types are useful when you want to operate on something but don't care about the details of the type in it.
- The forSome keyword is used to define existential types in Scala

```
1   package ExistentialTypes
2
3   object ExistentialTypesExample3 {
4     // The forSome keyword is used to define existential types in Scala.
5     def getLength(x : Array[T] forSome { type T}): Int = x.length
6
7     val stringArray = Array[String]("foo", "bar", "baz")
8
9     getLength(stringArray)
10  }
```

5. Type classes

- Type classes are a powerful concept that is heavily used in functional programming. They were first introduced in Haskell to achieve ad-hoc polymorphism
- A type class is a group of types that satisfy a contract typically defined by a trait
- A type class is a type system construct that supports ad hoc polymorphism. This is achieved by adding constraints to type variables in parametrically polymorphic types. Such a constraint typically involves a type class T and a type variable a, and means that a can only be instantiated to a type whose members support the overloaded operations associated with T
- A method can have an *implicit* parameter list, marked by the *implicit* keyword at the start of the parameter list. If the parameters in that parameter list are not passed as usual, Scala will look if it can get an implicit value of the correct type, and if it can, pass it automatically.