L14: Dependency Injection

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

February 27th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Recap from Tuesday

- We talked about the type system
- Upperbound
  - Think of this as the “ceiling” for the type
- Lowerbound
  - Think of this as the “floor” for the type
- Parameterizing types A = 5

  C[A] = Type A
- Variance
  - Invariance - Strick equal
  - Covarience (the + sign you put in) - Bigger than
  - Contravarience (the – sign you put in) - Same or smaller

Dependency

- What is a dependency?
- Why can this be useful? - Can put some constraint, declare implement afterward

- Can we manually "insert" this dependency?

Dependency Injection

- You might have heard this from software engineering
  - One object supplies the dependent item to another

- In this case, we can decouple the actual implementation away from the abstraction

trait A

— extends A

— extends A

- Let's use an example

Dependency Injection: Use Cases

Create bound between con and actual database name

Database name

- Let's assume class X needs a database connection
  - val con = DBConnectionRepository.getByName("appDBConnection")
- Notice this creates the dependency, but the dependency is coupled to the repository
  - Basically you need the repo, and cannot really change the name

- Dependency injection aims to decouple this dependency
  - You can make the connection declaration
  - Then you can implement the repository later

What About Interitance/Interface?

- Q: Why not just keep extending the traits to covers possible?
- trait FooAble {
  def foo = "this is an ordinary foo“
  }
- What if I create a code that depends on FooAble
- class BarUsingFooAble extends FooAble {
  def bar = "bar calls foo: " + foo => bar calls foo: this is an ...
  }
- object Main extends App {
  val barWithFooAble = new BarUsingFooAble
  println(barWithFooAble.bar)
  }
- What is the problem here?

↳ Can't change behavior of FooAble, BarUsingFooAble

You Cannot Do This (Can't Compile)

- Using the "with" keyword
  - What is the "with" keyword?
- class BarUsingFooAble {
  def bar = "bar calls foo: " + foo
  }
- object Main extends App {
  val barWithFooAble = new BarUsingFooAble with FooAble
  println(barWithFooAble.bar)
  }
  - This is ''done'' at instantiation
- Why this does not compile?

↳ Don't know foo, because it doesn't extend FooAble

What About Abstract Method?

trait FooAble
def foo

- abstract class BarUsingFooAble {
  def foo: FooAble
  def bar = "bar calls foo: " + foo.foo
  }
- object Main extends App {
  val fooInstance = new FooAble {}
  val barWithFoo = new BarUsingFooAble {
  def foo = fooInstance
  }
  println(barWithFoo.bar)
  }
- This gets messy as you extends

Baking a Cake

- We can decouple the dependency using the cake method
  - What?
- Self-type annotation
- trait FooAble { (Interface)

  def foo = "this is an ordinary foo“

  }
- class BarUsingFooAble { (This class will use FooAble interface)

  this: FooAble =>

  def bar = "bar calls foo: " + foo - have same scope as FooAble

  (Whatever we define in trait FooAble)

  }

- Anything after the => can use ***methods and variables of FooAble***

class/object: trait name =>

Baking a Cake

- With the [trait] => …
  - We declare that the class depends on [trait]
- Difference between this and “extends”:
  - extends is very type specific
    - You need to strictly have that exact type
  - Self-type annotation just say “I am declaring that whatever goes in the … will extend the trait”
    - It does not actually extend it yet
    - But it needs to conforms to [trait] (i.e., FooAble from the earlier example)
    - Hence, baking

Example

- trait FooService {
  def foo: String
  }
- **trait DefaultFoo extends FooService {**
  def foo = "default foo“
  }
  trait LuxuriousFoo extends FooService {
  def foo = "exclusive foo“
  }

[String]

- class BarUsingFooService {
  this: FooService => def bar = "bar uses foo: " + foo
  }
- object Main extends App {
  val barWithDefaultFoo = new BarUsingFooService with DefaultFoo
  val barWithBetterFoo = new BarUsingFooService with LuxuriousFoo
  println(barWithDefaultFoo.bar)
  println(barWithBetterFoo.bar)
- }

In-Class Exercise 13

- Please check our Canvas for the skeleton code

- We will try to implement a random draw using dependency injection

Out-of-class Exercise 14

- There is only one more Scala-based class
- Go through the “Why Rust” document
- Play around with Rust a bit so that you can do
  - Printing things
  - Running basic operations
  - Call functions
  - Use loop/conditionals