Lecture 10 – Contextual Abstractions

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

- **Advanced type systems**
  - Intersection and Union Types
  - Self Types
  - Opaque Types
  - Structural Types
  - Type Lambdas
  - Polymorphic Function Types
  - Match Types

Contextual Abstractions

Contextual Abstractions are a way to abstract over the context.

They are all variations of **term inference**; for a given type, the compiler automatically **infers** a **term** that has that type.

Other languages have been **influenced by Scala** in this regard.

- **Rust**'s traits or **Swift**'s protocol extensions
- **Design proposals** for other languages are also on the table:
  - for **Kotlin** as compile time dependency resolution
  - for **C#** as Shapes and Extensions
  - for **F#** as Traits
- Also a common feature of **theorem provers** such as Coq or Agda

Contents

Contents

Context Parameters

Assume that we want to define a method that differently renders the content of a website depending on its configuration.

```
case class Html(body: List[String])
case class Config(bgColor: String, color: String)

def renderHtml(html: Html, config: Config): String =
  renderBody(html.body, config)
def renderBody(body: List[String], config: Config): String =
  body.map(renderElem(_, config)).mkString("\n")
def renderElem(elem: String, config: Config): String =
  val Config(bgColor, color) = config
  s"<p style='background: $bgColor; color: $color'>$elem</p>"
```

```
renderHtml(Html(List("A", "B", "C")), Config("red", "yellow"))
renderHtml(Html(List("D", "E", "F")), Config("blue", "green"))
```

However, it has a **drawback**: we need to pass the `config` parameter to **every method** that **needs** it.

Context Parameters

Context parameters defined by `using` keyword make us able to not explicitly pass the `config` parameter to every method that needs it.

```
case class Html(body: List[String])
case class Config(bgColor: String, color: String)

def renderHtml(html: Html)(using config: Config): String =
  renderBody(html.body)  // no need to pass `config`
def renderBody(body: List[String])(using config: Config): String =
  body.map(renderElem).mkString("\n")  // no need to pass `config`
def renderElem(elem: String)(using config: Config): String =
  val Config(bgColor, color) = config
  s"<p style='background: $bgColor; color: $color'>$elem</p>"
```

```
renderHtml(Html(List("A", "B", "C")))(using Config("red", "yellow"))
renderHtml(Html(List("D", "E", "F")))(using Config("blue", "green"))
```

We can provide **contextual arguments** using `using` keyword.

Context Parameters

If we do **not need to refer** to the config parameter in the method body, we can even **omit the parameter name**:

```
case class Html(body: List[String])
case class Config(bgColor: String, color: String)

def renderHtml(html: Html)(using Config): String =
  renderBody(html.body)  // no need to pass `config`
def renderBody(body: List[String])(using Config): String =
  body.map(renderElem).mkString("\n")  // no need to pass `config`
def renderElem(elem: String)(using config: Config): String =
  val Config(bgColor, color) = config
  s"<p style='background: $bgColor; color: $color'>$elem</p>"
```

```
renderHtml(Html(List("A", "B", "C")))(using Config("red", "yellow"))
renderHtml(Html(List("D", "E", "F")))(using Config("blue", "green"))
```

Context Parameters – Given Instances

If we want to use a **single instance** for a particular type in the current context, we can define a **given instance** for that type.

```
given Config = Config("red", "yellow")
```

Then, we can call `renderHtml` by implicitly passing the given instance:

```
// implicitly pass `Config("red", "yellow")` to a context parameter
renderHtml(Html(List("A", "B", "C")))
```

We can define **multiple given instances** for the same type with names:

```
given config1: Config = Config("red", "yellow")
given config2: Config = Config("blue", "green")

renderHtml(Html(List("A", "B", "C")))(using config1)
renderHtml(Html(List("D", "E", "F")))(using config2)
```

Context Parameters – Exampels

Let's define a `Circle` class as follows:

```
case class Circle(radius: Double)
```

Assume that we want to define a method to magnify a circle by a given factor without modifying the `Circle` class.

```
def magnify(c: Circle)(using k: Int): Circle = Circle(c.radius * k)
```

Then, we can call it by explicitly passing the magnification factor:

```
magnify(Circle(3.0))(using 2)        // Circle(6.0)
```

or by defining a **given instance** for the `Int` type:

```
given magnifier: Int = 2
magnify(Circle(3.0))                 // Circle(6.0)
```

Contents

Implicit Conversions

In general, programming languages have a **fixed set of implicit conversions** that are built into the language.

However, Scala allows users to define their own **implicit conversions** by defining **given instances** for the `Conversion` type.

For example, we can define an **implicit conversion** from `String` to `Int` as its length with a **given instance** for the `Conversion[String, Int]` type:

```
given Conversion[String, Int] = (s: String) => s.length
```

Then, Scala compiler automatically converts `String` to `Int` when needed:

```
val len: Int = "hello"  // implicitly converted to 5
```

We can give a name to the given instance:

```
given stringToInt: Conversion[String, Int] = (s: String) => s.length
```

Implicit Conversions – Example

Assume that we have Circle and Square classes as follows:

```
case class Circle(radius: Double)
case class Square(side: Double)
```

Let's define a **implicit conversion** from Circle to Square that converts a circle to a square with the **same area**:

```
given circleToSquare: Conversion[Circle, Square] =
  (c: Circle) => Square(math.sqrt(math.Pi * c.radius * c.radius))
```

Scala compiler automatically converts Circle to Square when needed:

```
val square: Square = Circle(3.0)
// implicitly converted to Square(5.317361552716548)
// because sqrt(9 * Pi) = sqrt(28.274333882308138) = 5.317361552716548
```

Implicit Conversions – Example

However, Scala does not support **chained implicit conversions**.

```scala
case class Circle(radius: Double)
case class Square(side: Double)
type Area = Double

given c2a: Conversion[Circle, Area] = c => math.Pi * c.radius * c.radius
given a2s: Conversion[Area, Square] = a => Square(math.sqrt(a))

val area: Area = Circle(3.0)
val square1: Square = area
val square2: Square = Circle(3.0) // error: no implicit conversion found
```

We need to define an **implicit conversion** from `Circle` to `Square`:

```scala
given Conversion[Circle, Square] = c => a2s(c2a(c))

val square2: Square = Circle(3.0) // Square(5.317361552716548)
```

Contents

PLRG

Extension Methods

Imagine someone else defined a `Circle` class as follows:

```
case class Circle(radius: Double)
```

Now, assume that we want to define a method to calculate the area of a circle without modifying the `Circle` class.

Then, we need to define a top-level method `area` as follows:

```
def getArea(c: Circle): Double = math.Pi * c.radius * c.radius
```

Now, we can call this method as follows:

```
val circle: Circle = Circle(3.0)
getArea(circle)    // 9 * Pi = 28.274333882308138
```

Extension Methods

On the other hand, **extension methods** let us **add new methods** to a type without modifying the type definition.

```
extension (c: Circle)
  def area: Double = math.Pi * c.radius * c.radius
```

In this code,

- `Circle` is the type that the extension method is added to.
- The `c: Circle` syntax lets you refer to the variable `c` in your extension method.

We can call the method `area` as if it were a method of the `Circle` class:

```
val circle: Circle = Circle(3.0)
circle.area    // 9 * Pi = 28.274333882308138
```

Extension Methods

We can even define extension methods for **Scala built-in types**, including primitive types, such as `Int`:

```
extension (n: Int)
  def isEven: Boolean = n % 2 == 0

42.isEven  // true
3.isEven   // false
```

We can define multiple extension methods for the same type:

```
extension (n: Int)
  def square: Int = d * d
  def cube: Int = d * d * d

3.square        // 3 * 3     = 9
3.cube          // 3 * 3 * 3 = 27
2.square.cube   // 2 * 2     = 4   and    4 * 4 * 4 = 64
```

Contents

Given Imports

We defined **given instances** in the object `A`:

```
case class Circle(radius: Double)
case class Square(side: Double)
object A:
  given magnifier: Int = 2
  given circleToSquare: Conversion[Circle, Square] =
    (c: Circle) => Square(math.sqrt(math.Pi * c.radius * c.radius))
```

How to **import** these given instances in another object `B`?

```
object B:
  import A.{magnifier, circleToSquare}

  def magnify(c: Circle)(using k: Int): Circle = Circle(c.radius * k)

  // passing `magnifier` implicitly to `k` for `magnify`
  // implicitly converting `Circle` to `Square`
  val square: Square = magnify(Circle(3.0))
```

Given Imports

Note that `import A.*` imports **all non-given members** in `A`:

```
object B:
  import A.*      // import all non-given members in `A`
                  // not importing `magnifier` and `circleToSquare`
  ...
```

To import **all given members** in `A`, we need to use `import A.given`:

```
object B:
  import A.given // import all given members in `A`,
                 // including `magnifier` and `circleToSquare`
  ...
```

Thus, to import **all member** no matter if they are given or not, we can use `import A.{*, given}`.

```
object B:
  import A.{*, given} // import all members in `A`
  ...
```

Contents

Type Classes

A **type class** is a well-known type system in functional programming that allows us to define a set of operations that can be applied to a type.

In Scala, we can define a **type class** using a **trait**.

For example, let's define a **type class** `Show[A]` that provides an **abstract extension method** `show` to convert an instance of type `A` to a `String`:

```
trait Show[A]:
  extension (a: A) def show: String
```

Consider the following `Person` class:

```
case class Person(name: String, age: Int)
```

Then, we can define a given instance for the `Show[Person]` type class:

```
given Show[Person] with
  extension (p: Person)
    def show: String = s"${p.firstName} (age: ${p.lastName})"
```

Type Classes – Context Bounds

PLRG

We can use the `Show[A]` type class as follows:

```
val person: Person = Person("Ryu", 52)
person.show  // "Ryu (age: 52)"
```

Let's define a method to convert a list of persons to a list of strings using the `Show[A]` type class:

```
def showAll[A](as: List[A])(using Show[A]): List[String] =
  as.map(_.show)

val persons = List(Person("Ryu", 52), Person("Park", 32))
showAll(persons)  // List("Ryu (age: 52)", "Park (age: 32)")
```

We can simplify the method signature using a **context bound**:

```
def showAll[A: Show](as: List[A]): List[String] = as.map(_.show)
```

A **context bound** `[A: Show]` is a shorthand syntax for expressing the pattern of a **context parameter** applied to a **type parameter**.

Type Classes – Examples

Scala supports `Ordering[A]` as a **built-in type class** for comparing instances of type `A`.

For example, we need to define a given instance for the `Ordering[A]` to use specific methods (e.g., `max`, `min`, `sorted`, etc) for `List[A]`:

```
val nums: List[Int] = List(3, 1, 5, 6, 2, 4)
nums.max    // 3
nums.min    // 1
nums.sorted // List(1, 2, 3, 4, 5, 6)
```

We can above methods because there is a given instance for the `Ordering[Int]` is already defined in the Scala standard library.

However, if we want to use above methods for a custom type, we need to define a given instance for the `Ordering[A]` type class.

Type Classes – Examples

```scala
case class Person(name: String, age: Int)
```

Let's define a **type class** `Ordering[A]` for the `Person` type:

```scala
given Ordering[Person] = Ordering.by((p: Person) => (p.age, p.name))
```

It means that we want to compare `Person` instances by their ages but if the ages are the same, we want to compare them by their names.

```scala
val ps = List(Person("A",3),Person("B",1),Person("C",7),Person("D",3))
ps.max      // Person(C, 7)
ps.min      // Person(B, 1)
ps.sorted   // List(Person(B,1),Person(A,3),Person(D,3),Person(C,7))
```

Summary

1. Context Parameters
2. Implicit Conversions
3. Extension Methods
4. Given Imports
5. Type Classes

Next Lecture

- Metaprogramming

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr