EPFL

Implicit Conversions

Effective Programming in Scala

Implicit Conversions

Implicit conversions make it possible to convert an expression to a different type.

This mechanism is usually used to provide more ergonomic APIs.

Implicit Conversions

Implicit conversions make it possible to convert an expression to a different type.

This mechanism is usually used to provide more ergonomic APIs.

Example: API for defining JSON documents.

```
// { ”name”: ”Paul”, ”age”: 42 }
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

Modeling JSON Values

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json
```

Modeling JSON Values

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json

// { ”name”: ”Paul”, ”age”: 42 }
JObject(”name” -> JString(”Paul”), ”age” -> JNumber(42))
```

Aside: Repeated Parameters

```
def printSquares(xs: Int*) = println(xs.map(x => x * x))
printSquares(1, 2, 3) // ”Seq(1, 4, 9)”
```

- xs is a **repeated** parameter,
- at call site, we can supply several arguments,
- in the method body, xs has type Seq[Int],
- repeated parameters can only appear at the end of a parameter list.

Type Coercion: Motivation (1)

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json

// { ”name”: ”Paul”, ”age”: 42 }
JObject(”name” -> JString(”Paul”), ”age” -> JNumber(42))
```

Problem: Constructing JSON objects is too verbose.

Type Coercion: Motivation (2)

```
sealed trait Json
case class JNumber(value: BigDecimal) extends Json
case class JString(value: String) extends Json
case class JBoolean(value: Boolean) extends Json
case class JArray(elems: List[Json]) extends Json
case class JObject(fields: (String, Json)*) extends Json

// { ”name”: ”Paul”, ”age”: 42 }
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

How could we support the above user-facing syntax?

Type Coercion: Motivation (3)

```
// { ”name”: ”Paul”, ”age”: 42 }
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

What could be the type signature of the `obj` constructor?

(String, Any)*

Type Coercion: Motivation (3)

```
// { ”name”: ”Paul”, ”age”: 42 }
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

What could be the type signature of the `obj` constructor?

```
def obj(fields: (String, Any)*): Json
```

Type Coercion: Motivation (3)

```
// { ”name”: ”Paul”, ”age”: 42 }
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

What could be the type signature of the `obj` constructor?

```
def obj(fields: (String, Any)*): Json
```

Allows invalid JSON objects to be constructed!

```
Json.obj(”name” -> ((x: Int) => x + 1))
```

We want invalid code to be signaled to the programmer with a compilation error.

Type Coercion (1)

```
object Json:

  def obj(fields: (String, JsonField)*): Json =
    JObject(fields.map(_.json)*)

  case class JsonField(json: Json)

end Json
```

Type Coercion (2)

```
case class JsonField(json: Json)

object JsonField:
  given fromString: Conversion[String, JsonField] with
    def apply(s: String) = JsonField(JString(s))
  given fromInt: Conversion[Int, JsonField] with
    def apply(n: Int) = JsonField(JNumber(n))
  ...
  given fromJson: Conversion[Json, JsonField] with
    def apply(j: Json) = JsonField(j)
```

Type Coercion: Usage

To be able to use implicit conversions, we have to inform the compiler of our intent by writing the import clause `import scala.language.implicitConversions`.

```
import scala.language.implicitConversions

Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

Type Coercion: Usage

To be able to use implicit conversions, we have to inform the compiler of our intent by writing the import clause `import scala.language.implicitConversions`.

```
import scala.language.implicitConversions
```

```
Json.obj(”name” -> ”Paul”, ”age” -> 42)
```

The compiler implicitly inserts the following conversions:

```
Json.obj(
  ”name” -> Json.JsonField.fromString.apply(”Paul”),
  ”age” -> Json.JsonField.fromInt.apply(42)
)
```

Implicit Conversions

The compiler looks for implicit conversions on an expression `e` of type `T` if `T` does not conform to the expression's expected type `S`.

In such a case, the compiler looks in the context for a given instance of type `Conversion[T, S]`.

Note: at most one implicit conversion can be applied to a given expression.

Warning

Implicit conversions are silently applied by the compiler, and they change the type of expressions.

Therefore, they can confuse developers reading code (hence the required import).

Before defining an implicit conversion, make sure to weigh the pros and cons. Reducing boilerplate is a good purpose, but this should be balanced with the possible drawbacks of not seeing pieces of code that are yet part of the program.

Summary

- Implicit conversions can improve the ergonomics of an API but should be used *sparingly*