Implicit Conversions

- The last implicit-related mechanism of the language is implicit **conversions**
- They make it possible to automatically convert an expression of a given type to an expression of a different type
- This mechanism is usually used to provide more ergonomic APIs

Type Coercion

- **JSON** is a data-interchange format often used in web applications. As an example, here is a JSON document describing a user with a name and an age:

  { "name": "Paul", "age": 42 }
- JSON is text-based format. To manipulate JSON documents in our programs, it is more convenient to parse these documents into an **Abstract Syntax Tree(AST)**
- Here is a Scala definition of a Json AST:

  ```
  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
  ```
- With this definition, the JSON document shown above can be constructed like the following: ***JObject("name" -> JString("Paul"), "age" -> JNumber(42))***
- This works fine, but it is a bit more verbose than the plain JSON syntax. Would it be possible to design an API providing a syntax closer to plain JSON than the current syntax? Would it be possible to support the following user-facing Scala syntax:

  obj("name" -> "Paul", "age" -> 42)
- The challenge is that values of object fields can have various types: in this example, the name field is a string, whereas the age field is a number. As a consequence, the obj constructor should accept both String and Int field values.
- A naive solution could be to accept a field value of type Any:

  def obj(fields: (String, Any)*): Json
- However, this solution allows invalid JSON objects to be constructed:

  obj("name" -> ((x: Int) => x + 1)) *// INVALID!*
- Instead, we want invalid constructions to be rejected with a compilation error
- To support the desired syntax without making compromises on type safety, the obj constructor has to take values of type Json:

  def obj(fields: (String, Json)*): Json = JObject(fields: _*)
- However, we want users to be able to just write obj("foo" -> "bar") instead of obj("foo" -> JString("bar")) and so on for other types of values that have a direct representation in JSON

- To achieve that, we provide **implicit conversions** to convert String values to JSON string values, and Int values to JSON number values, and so on for other possible JSON values:

```
object Json {
  import scala.language.implicitConversions
  implicit def stringToJson(s: String): Json = JString(s)
  implicit def intToJson(n: Int): Json = JNumber(n)
  …
}
```

- Before defining the implicit conversions, we inform the compiler of our intent by writing the import clause **import scala.language.implicitConversions**
- Implicit conversions are implicit definitions that take exactly one (non-implicit) parameter. These conversions can be used by the compiler when the code would otherwise not type check
- Now, all pieces are in place to support the desired user-facing syntax:

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

- This expression is not well typed as it is written, so the compiler implicitly inserts the following conversions to make it well typed:

```
obj(
  "name" -> Json.stringToJson("Paul"),
  "age" -> Json.intToJson(42)
)
```

- This example shows that implicit conversions can be used to perform **type coercion**

Extension Methods

- The second example shows how to implement extension methods
- Consider the following type Duration:

```
case class Duration(value: Int, unit: TimeUnit)
```

- Defining a duration of “15 seconds” looks like the following:

```
val delay = Duration(15, TimeUnit.Second)
```

- Would it be possible to support a more concise and direct syntax, like this one for instance: ***val delay = 15.seconds***
- We need to enrich the type Int with a method seconds. We achieve this by creating an **implicit class** HasSeconds that has a method seconds and that takes a number of seconds as a constructor parameter:

```
case class Duration(value: Int, unit: TimeUnit)

object Duration {
  object Syntax {
    import scala.language.implicitConversions

    implicit class HasSeconds(n: Int) {
      def seconds: Duration = Duration(n, TimeUnit.Second)
    }
  }
}
```

- The usage looks like this:
  import Duration.Sytax._
  val delay = 15.seconds
- The compiler implicitly inserts the following conversion:
  val delay = new HasSeconds(15).seconds
- Constructors of implicit classes act as implicit conversions

Implicit Conversions

- We have seen two examples of usage of implicit conversions. In this section, we explain how the compiler uses (and doesn't use) them
- The compiler looks for implicit conversions on an expression e of type T in the following situations:
  - T does not conform to the expression's expected type
  - In a selection e.m, if member m is not accesbile on T
  - In a selection e.m(args), if member m is accessible on T, but is not applicable to the arguments args
- In all these cases, the compiler will look for an implicit conversion that, if it is applied, makes the expression type check
- Implicit conversions are implicit methods (or implicit class constructors) that take exactly one non-implicit parameter (and possibly additional implicit parameters)
- Note: at most one implicit conversion can be applied to a given expression
- Implicit conversions are searched in the same places as implicit definitions: in the enclosing lexical scope(local definitions, inherited definitions or imported definitions) or in the implicit search scope(companion objects) of the expected type or the expression's type
- For instance, in the second example the implicit class HasSeconds is found in the imported implicit definitions(import Duration.Syntax._). However, in the first example, the implicit conversions are found in the companion object of the expected type Json(the method obj expects arguments of type Json)
- We conclude this lesson with a warning. Because implicit conversions are silently applied by the compiler and they change the type of expressions, they can confuse developers reading code. Care must be taken when using implicit conversions: reducing boilerplate is a good purpose, but this should always be balanced with the possible drawbacks of not seeing pieces of code that are yet part of the program