Implicit Conversions

Implicit conversions are defined by given instances of the `scala.Conversion` class. This class is defined in package `scala` as follows:

```
abstract class Conversion[-T, +U] extends (T => U):
  def apply (x: T): U
```

For example, here is an implicit conversion from `String` to `Token`:

```
given Conversion[String, Token] with
  def apply(str: String): Token = new KeyWord(str)
```

Using an alias this can be expressed more concisely as:

```
given Conversion[String, Token] = new KeyWord(_)
```

An implicit conversion is applied automatically by the compiler in three situations:

1. If an expression `e` has type `T`, and `T` does not conform to the expression's expected type `S`.
2. In a selection `e.m` with `e` of type `T`, but `T` defines no member `m`.
3. In an application `e.m(args)` with `e` of type `T`, if `T` does define some member(s) named `m`, but none of these members can be applied to the arguments `args`.

In the first case, the compiler looks for a given `scala.Conversion` instance that maps an argument of type `T` to type `S`. In the second and third case, it looks for a given `scala.Conversion` instance that maps an argument of type `T` to a type that defines a member `m` which can be applied to `args` if present. If such an instance `C` is found, the expression `e` is replaced by `C.apply(e)`.

Examples

1. The `Predef` package contains "auto-boxing" conversions that map primitive number types to subclasses of `java.lang.Number`. For instance, the conversion from `Int` to `java.lang.Integer` can be defined as follows:

```
given int2Integer: Conversion[Int, java.lang.Integer] =
  java.lang.Integer.valueOf(_)
```

2. The "magnet" pattern is sometimes used to express many variants of a method. Instead of defining overloaded versions of the method, one can also let the method take one or more arguments of specially defined "magnet" types, into which various argument types can be converted. Example:

```
object Completions:

  // The argument "magnet" type
  enum CompletionArg:
    case Error(s: String)
    case Response(f: Future[HttpResponse])
    case Status(code: Future[StatusCode])

  object CompletionArg:

    // conversions defining the possible arguments to pass to `complete`
    // these always come with CompletionArg
    // They can be invoked explicitly, e.g.
    //
    // CompletionArg.fromStatusCode(statusCode)

    given fromString    : Conversion[String, CompletionArg]
    given fromFuture    : Conversion[Future[HttpResponse], CompletionArg]
    given fromStatusCode: Conversion[Future[StatusCode], CompletionArg]
  end CompletionArg
  import CompletionArg.*

  def complete[T](arg: CompletionArg) = arg match
    case Error(s) => ...
    case Response(f) => ...
    case Status(code) => ...

end Completions
```

This setup is more complicated than simple overloading of `complete`, but it can still be useful if normal overloading is not available (as in the case above, since we cannot

have two overloaded methods that take `Future[...]` arguments), or if normal overloading would lead to a combinatorial explosion of variants.

Contex... By-Na...

Contributors to this page

 pikinier20  BarkingBad  julienrf  odersky  michelou

 smarter