Extension Methods

Extension methods allow one to add methods to a type after the type is defined. Example:

```
case class Circle(x: Double, y: Double, radius: Double)

extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2
```

Like regular methods, extension methods can be invoked with infix `.`:

```
val circle = Circle(0, 0, 1)
circle.circumference
```

Translation of Extension Methods

An extension method translates to a specially labelled method that takes the leading parameter section as its first argument list. The label, expressed as `<extension>` here, is compiler-internal. So, the definition of `circumference` above translates to the following method, and can also be invoked as such:

```
<extension> def circumference(c: Circle): Double = c.radius * math.Pi * 2

assert(circle.circumference == circumference(circle))
```

Operators

The extension method syntax can also be used to define operators. Examples:

```
extension (x: String)
  def < (y: String): Boolean = ...
extension (x: Elem)
  def +: (xs: Seq[Elem]): Seq[Elem] = ...
```

```
extension (x: Number)
  infix def min (y: Number): Number = ...


"ab" < "c"
1 +: List(2, 3)
x min 3
```

The three definitions above translate to

```
<extension> def < (x: String)(y: String): Boolean = ...
<extension> def +: (xs: Seq[Elem])(x: Elem): Seq[Elem] = ...
<extension> infix def min(x: Number)(y: Number): Number = ...
```

Note the swap of the two parameters `x` and `xs` when translating the right-associative operator `+:` to an extension method. This is analogous to the implementation of right binding operators as normal methods. The Scala compiler preprocesses an infix operation `x +: xs` to `xs.+:(x)`, so the extension method ends up being applied to the sequence as first argument (in other words, the two swaps cancel each other out). See here for details.

Generic Extensions

It is also possible to extend generic types by adding type parameters to an extension. For instance:

```
extension [T](xs: List[T])
  def second = xs.tail.head

extension [T: Numeric](x: T)
  def + (y: T): T = summon[Numeric[T]].plus(x, y)
```

Type parameters on extensions can also be combined with type parameters on the methods themselves:

```
extension [T](xs: List[T])
  def sumBy[U: Numeric](f: T => U): U = ...
```

Type arguments matching method type parameters are passed as usual:

```
List("a", "bb", "ccc").sumBy[Int](_.length)
```

By contrast, type arguments matching type parameters following `extension` can be passed only if the method is referenced as a non-extension method:

```
sumBy[String](List("a", "bb", "ccc"))(_.length)
```

Or, when passing both type arguments:

```
sumBy[String](List("a", "bb", "ccc"))[Int](_.length)
```

Extensions can also take using clauses. For instance, the `+` extension above could equivalently be written with a using clause:

```
extension [T](x: T)(using n: Numeric[T])
  def + (y: T): T = n.plus(x, y)
```

Collective Extensions

Sometimes, one wants to define several extension methods that share the same left-hand parameter type. In this case one can "pull out" the common parameters into a single extension and enclose all methods in braces or an indented region. Example:

```
extension (ss: Seq[String])

  def longestStrings: Seq[String] =
    val maxLength = ss.map(_.length).max
    ss.filter(_.length == maxLength)

  def longestString: String = longestStrings.head
```

The same can be written with braces as follows (note that indented regions can still be used inside braces):

```
extension (ss: Seq[String]) {

  def longestStrings: Seq[String] = {
    val maxLength = ss.map(_.length).max
    ss.filter(_.length == maxLength)
  }

  def longestString: String = longestStrings.head
}
```

Note the right-hand side of `longestString`: it calls `longestStrings` directly, implicitly assuming the common extended value `ss` as receiver.

Collective extensions like these are a shorthand for individual extensions where each method is defined separately. For instance, the first extension above expands to

method is defined separately. For instance, the first extension above expands to:

```
extension (ss: Seq[String])
  def longestStrings: Seq[String] =
    val maxLength = ss.map(_.length).max
    ss.filter(_.length == maxLength)

extension (ss: Seq[String])
  def longestString: String = ss.longestStrings.head
```

Collective extensions also can take type parameters and have using clauses. Example:

```
extension [T](xs: List[T])(using Ordering[T])
  def smallest(n: Int): List[T] = xs.sorted.take(n)
  def smallestIndices(n: Int): List[Int] =
    val limit = smallest(n).max
    xs.zipWithIndex.collect { case (x, i) if x <= limit => i }
```

Translation of Calls to Extension Methods

To convert a reference to an extension method, the compiler has to know about the extension method. We say in this case that the extension method is *applicable* at the point of reference. There are four possible ways for an extension method to be applicable:

1. The extension method is visible under a simple name, by being defined or inherited or imported in a scope enclosing the reference.
2. The extension method is a member of some given instance that is visible at the point of the reference.
3. The reference is of the form `r.m` and the extension method is defined in the implicit scope of the type of `r`.
4. The reference is of the form `r.m` and the extension method is defined in some given instance in the implicit scope of the type of `r`.

Here is an example for the first rule:

```
trait IntOps:
  extension (i: Int) def isZero: Boolean = i == 0

  extension (i: Int) def safeMod(x: Int): Option[Int] =
    // extension method defined in same scope IntOps
    if x.isZero then None
    else Some(i % x)
```

```
object IntOpsEx extends IntOps:
  extension (i: Int) def safeDiv(x: Int): Option[Int] =
    // extension method brought into scope via inheritance from IntOps
    if x.isZero then None
    else Some(i / x)

trait SafeDiv:
  import IntOpsEx.* // brings safeDiv and safeMod into scope

  extension (i: Int) def divide(d: Int): Option[(Int, Int)] =
    // extension methods imported and thus in scope
    (i.safeDiv(d), i.safeMod(d)) match
      case (Some(d), Some(r)) => Some((d, r))
      case _ => None
```

By the second rule, an extension method can be made available by defining a given instance containing it, like this:

```
given ops1: IntOps()  // brings safeMod into scope

1.safeMod(2)
```

By the third and fourth rule, an extension method is available if it is in the implicit scope of the receiver type or in a given instance in that scope. Example:

```
class List[T]:
  ...
object List:
  ...
  extension [T](xs: List[List[T]])
    def flatten: List[T] = xs.foldLeft(List.empty[T])(_ ++ _)

  given [T: Ordering]: Ordering[List[T]] with
    extension (xs: List[T])
      def < (ys: List[T]): Boolean = ...
end List

// extension method available since it is in the implicit scope
// of List[List[Int]]
List(List(1, 2), List(3, 4)).flatten

// extension method available since it is in the given Ordering[List[T]],
// which is itself in the implicit scope of List[Int]
List(1, 2) < List(3)
```

The precise rules for resolving a selection to an extension method are as follows.

Assume a selection `e.m[Ts]` where `m` is not a member of `e`, where the type arguments `[Ts]` are optional, and where `T` is the expected type. The following two rewritings are tried in order:

1. The selection is rewritten to `m[Ts](e)`.
2. If the first rewriting does not typecheck with expected type `T`, and there is an extension method `m` in some eligible object `o`, the selection is rewritten to `o.m[Ts](e)`. An object `o` is *eligible* if
   - `o` forms part of the implicit scope of `T`, or
   - `o` is a given instance that is visible at the point of the application, or
   - `o` is a given instance in the implicit scope of `T`.

   This second rewriting is attempted at the time where the compiler also tries an implicit conversion from `T` to a type containing `m`. If there is more than one way of rewriting, an ambiguity error results.

An extension method can also be referenced using a simple identifier without a preceding expression. If an identifier `g` appears in the body of an extension method `f` and refers to an extension method `g` that is defined in the same collective extension

```
extension (x: T)
  def f ... = ... g ...
  def g ...
```

the identifier is rewritten to `x.g`. This is also the case if `f` and `g` are the same method. Example:

```
extension (s: String)
  def position(ch: Char, n: Int): Int =
    if n < s.length && s(n) != ch then position(ch, n + 1)
    else n
```

The recursive call `position(ch, n + 1)` expands to `s.position(ch, n + 1)` in this case. The whole extension method rewrites to

```
def position(s: String)(ch: Char, n: Int): Int =
  if n < s.length && s(n) != ch then position(s)(ch, n + 1)
  else n
```

Syntax

Here are the syntax changes for extension methods and collective extensions relative to the current syntax.

```
BlockStat          ::=  ... | Extension
TemplateStat       ::=  ... | Extension
TopStat            ::=  ... | Extension
Extension          ::=  'extension' [DefTypeParamClause] {UsingParamClause}
                        '(' DefParam ')' {UsingParamClause} ExtMethods
ExtMethods         ::=  ExtMethod | [nl] <<< ExtMethod {semi ExtMethod} >>>
ExtMethod          ::=  {Annotation [nl]} {Modifier} 'def' DefDef
```

In the above the notation `<<< ts >>>` in the production rule `ExtMethods` is defined as follows :

```
<<< ts >>>         ::=  '{' ts '}' | indent ts outdent
```

`extension` is a soft keyword. It is recognized as a keyword only if it appears at the start of a statement and is followed by `[` or `(`. In all other cases it is treated as an identifier.

‹ Importi... Right-... ›

Contributors to this page

 pikinier20  BarkingBad  julienrf  odersky som-snytt

 anatoliykmetyuk  ShapelessCat  michelou iroha168

 rjolly