Given Definitions

Effective Programming in Scala

Given Definitions

For the previous example to work, `Ordering.Int` must be a `given` definition:

```
object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int =
      if x < y then -1 else if x > y then 1 else 0
```

This code defines a given instance of type `Ordering[Int]`, named `Int`.

The instance is initialized the first time it is used.

Alias Givens

An alias can be used to define a given instance that is equal to some expression:

```
object IntOrdering extends Ordering[Int]:
  def compare(x: Int, y: Int): Int =
    if x < y then -1 else if x > y then 1 else 0

given intOrdering: Ordering[Int] = IntOrdering
```

Like with regular given instance definitions, the right-hand side of the alias is evaluated only once, the first time `intOrdering` is accessed, and this value is returned by all subsequent accesses.

Anonymous Given Definitions

Given definitions can be anonymous. Just omit their name:

```
given Ordering[Double] with
  def compare(x: Double, y: Double): Int =
    if x < y then -1 else if x > y then 1 else 0
```

The compiler will synthesize a name for an anonymous definition:

```
given given_Ordering_Double: Ordering[Double] with
  def compare(x: Double, y: Double): Int =
    if x < y then -1 else if x > y then 1 else 0
```

Complete Example

```
trait Ordering[A]:
  def compare(a1: A, a2: A): Int

object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int = ...
  given String: Ordering[String] with
    def compare(s: String, t: String): Int = ...

def sort[A](as: List[A])(using Ordering[A]): List[A] = ...

sort(List(1, 3, 2))           // : List[Int]    = List(1, 2, 3)
sort(List("banana", "apple")) // : List[String] = List("apple", "banana")
```

Summoning an Instance

One can refer to a (named or anonymous) instance by its type:

```
summon[Ordering[Int]]
summon[Ordering[Double]]
```

These expand to:

```
Ordering.Int
Ordering.given_Ordering_Double
```

`summon` is a predefined method. It can be defined like this:

```
def summon[T](using arg: T): T = arg
```

Context Parameters Resolution

Say, a function takes a context parameter of type `T`.

The compiler will search a given instance that:

- has a type compatible with `T`,
- is visible at the point of the method call, or is defined in a companion object *associated* with `T`.

If there is a single definition, it will be taken as actual arguments for the context parameter.

Otherwise, it's an error.

Visibility of Given Instances

```
class Foo

trait Givens:
  given Foo = Foo()

  summon[Foo] // OK, given Foo is defined in the enclosing scope

object Givens extends Givens:
  summon[Foo] // OK, given Foo is inherited from the trait Givens

summon[Foo] // Error: no given instances were found that match type Foo

import Givens.{given Foo}
summon[Foo] // OK, given Foo is imported
```

Importing Given Instances

Since given instances can be anonymous, how can they be imported?

In fact, there are three ways to import a given instance.

1. By-name:

```
import scala.math.Ordering.Int
```

2. By-type:

```
import scala.math.Ordering.{given Ordering[Int]}
import scala.math.Ordering.{given Ordering[?]}
```

3. With a `given` selector (imports all the given instances of a path):

```
import scala.math.Ordering.given
```

Since the names of givens don't really matter, the second form of import is preferred since it is most informative.

Given Instances Search Scope

The scope of a search for a given instance of type `T` includes:

- first, all the given instances that are visible (inherited, imported, or defined in any enclosing scope),
- then, the given instances found in any companion object *associated* with `T`.

The definition of *associated* is quite general. Besides the companion object of the type T itself, the compiler will also consider:

- companion objects associated with any of `T`'s inherited types,
- companion objects associated with any type argument in `T`,
- if `T` is an inner class, the outer objects in which it is embedded.

Companion Objects Associated With a Queried Type

The compiler searches first for a given instance matching the queried type in the lexical scope, and only then if no given instance is found does it search in associated companion objects.

Consider the following hierarchy:

```
trait Foo[A]
trait Bar[A] extends Foo[A]
trait Baz[A] extends Bar[A]
trait X
trait Y extends X
```

If a given instance of type `Bar[Y]` is required, the compiler will look into the companion objects `Bar`, `Y`, `Foo`, and `X` (but not `Baz`).

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

// : List[Int]

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

- In the enclosing scope
- Via a given import
- In a companion object associated with the type `Ordering[Int]`

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

o In the enclosing scope

o Via a given import

x In a companion object associated with the type `Ordering[Int]`

- The given instance is found in the `Ordering` companion object

No Given Instance Found

If there is no available given instance matching the queried type, a compilation error is reported:

```
summon[Int]

^^^^^^^^^^^
error: no implicit argument of type Int was found for
       parameter x of method summon
```

Ambiguous Given Instances

If more than one given instance is eligible, an **ambiguity** is reported:

```
given ord1: Ordering[Int] = Ordering.Int
given ord2: Ordering[Int] = Ordering.Int
summon[Ordering[Int]]

^^^^^^^^^^^^^^^^^^^^^
error: ambiguous implicit arguments: both method ord1
       and method org2 match type Ordering[Int] of
       parameter x of method summon
```

Summary

There has to be a **unique** given instance matching the queried type for it to be used by the compiler as a context argument.

Given instances are searched in the enclosing **lexical scope** (imports, parameters, inherited members) as well as in the **companion objects** of types *associated* with the queried type.