Contextual Abstraction

Principles of Functional Programming

Con-text

what comes with the text,
but is not in the text

Context Takes Many Forms

- the current configuration
- the current scope
- the meaning of "<" on this type
- the user on behalf of which the operation is performed
- the security level in effect
- ...

Code becomes more modular if it can *abstract* over context.

That is, functions and classes can be written without knowing in detail the context in which they will be called or instantiated.

How Is Context Represented?

So far:

- global values
- global mutable variables
- “Monkey Patching”
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables
- "Monkey Patching"
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- "Monkey Patching"
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- "Monkey Patching" *- more powerful ways to shoot yourself in the foot...*
- dependency injection frameworks (e.g. Spring, Guice)

How Is Context Represented?

So far:

- global values *i.e., no abstraction - this is often too rigid*
- global mutable variables *what if different modules need different settings? interference can be dangerous!*
- "Monkey Patching" *- more powerful ways to shoot yourself in the foot...*
- dependency injection frameworks (e.g. Spring, Guice) *- outside the language, rely on bytecode rewriting $\rightarrow$ harder to understand and debug.*

Functional Context Representation

In functional programming, the natural way to abstract over context is with function parameters.

+ flexible
+ types are checked
+ not relying on side effects

But sometimes this is too much of a good thing! It can lead to

– many function arguments
– which hardly ever change
– repetitive, errors are hard to spot

Example: Sorting

We have seen `sort` functions. For instance, here's an outline of a method `sort` that takes as parameter a `List[Int]` and returns another `List[Int]` containing the same elements, but sorted:

```
def sort(xs: List[Int]): List[Int] =
  ...
  ... if x < y then ...
  ...
```

At some point, this method has to compare two elements `x` and `y` of the given list.

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

But this does not work, because there's not a single comparison method `<` that works for all types.

Making `sort` more General

Problem: How to parameterize `sort` so that it can also be used for lists with elements other than `Int`, such as `Double` or `String`?

A straightforward approach would be to use a polymorphic type `T` for the type of elements:

```
def sort[T](xs: List[T]): List[T] = ...
```

But this does not work, because there's not a single comparison method `<` that works for all types.

In other words, we need to ask the question: What is the meaning of `<` on type `T` *at the call site*?

This means querying the call-site context.

Parameterization of `sort`

The most flexible design is to pass the comparison operation as an additional parameter:

```
def sort[T](xs: List[T])(lessThan: (T, T) => Boolean): List[T] =
  ...
  ... if lessThan(x, y) then ...
  ...
```

Calling Parameterized sort

We can now call sort as follows:

```
val ints = List(-5, 6, 3, 2, 7)
val strings = List("apple", "pear", "orange", "pineapple")

sort(ints)((x, y) => x < y)
sort(strings)((s1, s2) => s1.compareTo(s2) < 0)
```

Parameterization with Ordering

There is already a class in the standard library that represents orderings:

```
scala.math.Ordering[A]
```

Provides ways to compare elements of type `A`. So, instead of parameterizing with the `lessThan` function, we could parameterize with `Ordering` instead:

```
def sort[T](xs: List[T])(ord: Ordering[T]): List[T] =
  ...
  ... if ord.lt(x, y) then ...
  ...
```

Ordering Instances

Calling the new `sort` can be done like this:

```
import scala.math.Ordering

sort(ints)(Ordering.Int)
sort(strings)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

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

Reducing Boilerplate

Problem: Passing around `Ordering` arguments is cumbersome.

```
sort(ints)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` value always uses the same `Ordering.Int` argument, sorting a `List[String]` value always uses the same `Ordering.String` argument, and so on...

Implicit Parameters

We can reduce the boilerplate by making `ord` an *implicit parameter*.

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

Then, calls to `sort` can omit the `ord` parameter:

```
sort(ints)
sort(strings)
```

The compiler infers the argument value based on its expected type.

Type Inference

We have seen that the compiler is able to *infer types* from *values*.

That is, the previous calls to `sort` are augmented as follows:

```
sort(ints)
sort(strings)
```

Type Inference

We have seen that the compiler is able to *infer types* from *values*.

That is, the previous calls to `sort` are augmented as follows:

```
sort[Int](ints)
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (*aka* terms) from *types*.

When there is exactly one "obvious" value for a type in a `using` clause, the compiler can provide that value to us.

```
sort[Int](ints)
sort[String](strings)
```

Term Inference

The Scala compiler is also able to do the opposite, namely to *infer expressions* (*aka* terms) from *types*.

When there is exactly one "obvious" value for a type, the compiler can provide that value to us.

```
sort[Int](ints)(using Ordering.Int)
sort[String](strings)(using Ordering.String)
```

EPFL

Using Clauses and Given Instances

Principles of Functional Programming

Martin Odersky and Julien Richard-Foy

Using Clauses

An implicit parameter is introduced by a `using` parameter clause:

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

A matching explicit argument can be passed in a `using` argument clause:

```
sort(strings)(using Ordering.String)
```

But the argument can also be left out (and it usually is).

If the argument is missing, the compiler will infer one from the parameter type.

```
sort(strings)
```

Using Clauses Syntax Reference

Multiple parameters can be in a `using` clause:

```
def f(x: Int)(using a: A, b: B) = ...
f(x)(using a, b)
```

Or, there can be several `using` clauses in a row:

```
def f(x: Int)(using a: A)(using b: B) = ...
```

`using` clauses can also be freely mixed with regular parameters:

```
def f(x: Int)(using a: A)(y: Boolean)(using b: B) = ...
f(x)(using a)(y)(using b)
```

Anonymous Using Clauses

Parameters of a `using` clause can be anonymous:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] =
  ...
  ... merge(sort(fst), sort(snd))

def merge[T](xs: List[T])(using Ordering[T]): List[T] = ...
```

This is useful if the body of `sort` does not mention `ord` at all, but simply passes it on as an implicit argument to further methods.

Anonymous Using Clauses

Parameters of a `using` clause can be anonymous:

```
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] =
  ...
  ... merge(sort(fst), sort(snd))(using ord)

def merge[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

This is useful if the body of `sort` does not mention `ord` at all, but simply passes it on as an implicit argument to further methods.

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

Instead of:

```
def printSorted[T](as: List[T])(using Ordering[T]) =
  println(sort(as))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](as: List[T]) =
  println(sort(as))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](as: List[T]) =
  println(sort(as))
```

More generally, a method definition such as:

$$\textbf{def}\ f[T : U_1\ \ldots\ : U_n](ps) : R = ...$$

is expanded to:

$$\textbf{def}\ f[T](ps)(\textbf{using}\ U_1[T], \ldots, U_n[T]) : R = ...$$

Given Instances

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

```
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`.

Anonymous Given Instances

Given instances can be anonymous. Just omit the instance name:

```
given Ordering[Double] with
  def compare(x: Int, y: Int): Int = ...
```

The compiler will synthesize a name for an anonymous instance:

```
given given_Ordering_Double: Ordering[Double] with
  def compare(x: Int, y: Int): Int = ...
```

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 x: T) = x
```

Implicit Parameter Resolution

Say, a function takes an implicit 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 function call, or is defined in a companion object *associated* with `T`.

If there is a single (most specific) instance, it will be taken as actual arguments for the inferred parameter.

Otherwise it's an error.

Given Instances Search Scope

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

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

The definition of *associated* is quite general. Besides the companion object of a class 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

If the compiler does not find a given instance matching the queried type `T` in the lexical scope, it continues searching in the companion objects associated with `T`.

Consider the following hierarchy:

```
trait Foo[T]
trait Bar[T] extends Foo[T]
trait Baz[T] extends Bar[T]
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`).

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 wildcard:

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

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

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

o 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, an error is reported:

```
scala> def f(using n: Int) = ()
scala> f
       ^
error: no implicit argument of type Int was found for parameter n of method f
```

Ambiguous Given Instances

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

```
trait C:
  val x: Int
given c1: C with
  val x = 1
given c2: C with
  val x = 2

f(using c: C) = ()
f
^
error: ambiguous implicit arguments:
both value c1 and value c2
match type C of parameter c of method f
```

Priorities

Actually, several given instances matching the same type don't generate an ambiguity if one is *more specific* than the other.

In essence, a definition

```
given a: A
```

is more specific than a definition

```
given b: B
```

if:

- `a` is in a closer lexical scope than `b`, or
- `a` is defined in a class or object which is a subclass of the class defining `b`, or
- type `A` is a generic instance of type `B`, or
- type `A` is a subtype of type `B`.

Priorities: Example (1)

Which given instance is summoned here?

```
class A[T](x: T)
given universal[T](using x: T): A[T](x)
given specific: A[Int](2)

summon[A[Int]]
```

Priorities: Example (2)

Which given instance is summoned here?

```
trait A:
  given ac: C
trait B extends A:
  given bc: C
object O extends B:
  val x = summon[C]
```

Priorities: Example (3)

Which given instance is summoned here?

```
given ac: C
def f() =
  given b: C
  def g(using c: C) = ()

  g
```

Summary

In this lecture we have introduced a way to do *type-directed programming*, with the help of a language mechanism that infers *values* from *types*.

There has to be a *unique* (most specific) given instance matching the queried type for it to be used by the compiler.

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

Type Classes

Principles of Functional Programming

Martin Odersky and Julien Richard-Foy

Type Classes

In the previous lectures we have seen a particular pattern of code:

```
trait Ordering[A]:
  def compare(x: A, y: A): Int

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

Type Classes

We say that `Ordering` is a **type class**.

In Scala, a type class is a generic trait that comes with given instances for type instances of that trait.

E.g., in the `Ordering` example, we have given instances for `Ordering[Int]` and `Ordering[String]`

Type Classes

We say that `Ordering` is a **type class**.

In Scala, a type class is a generic trait that comes with given instances for type instances of that trait.

E.g., in the `Ordering` example, we have given instances for `Ordering[Int]` and `Ordering[String]`

Type classes provide yet another form of polymorphism:

The `sort` method can be called with lists containing elements of any type `A` for which there is a given instance of type `Ordering[A]`.

```
def sort[A: Ordering](xs: List[A]): List[A] = ...
```

At compilation-time, the compiler resolves the specific `Ordering` implementation that matches the type of the list elements.

Exercise

Implement an instance of the `Ordering` typeclass for the `Rational` type.

```
case class Rational(num: Int, denom: Int)
```

Reminder:

let $q = \frac{num_q}{denom_q}$, $r = \frac{num_r}{denom_r}$,

$q < r \Leftrightarrow \frac{num_q}{denom_q} < \frac{num_r}{denom_r} \Leftrightarrow num_q \times denom_r < num_r \times denom_q$

Digression: Retroactive Extension

It is worth noting that we were able to implement the `Ordering[Rational]` instance without changing the `Rational` class definition.

Type classes support *retroactive* extension: the ability to extend a data type with new operations without changing the original definition of the data type.

In this example, we have added the capability of comparing `Rational` numbers.

Conditional Instances

Question: How do we define an `Ordering` instance for lists?

Observation: This can be done only if the list elements have an ordering.

```
given listOrdering[A](using ord: Ordering[A]): Ordering[List[A]] with
```

Conditional Instances

Question: How do we define an `Ordering` instance for lists?

Observation: This can be done only if the list elements have an ordering.

```
given listOrdering[A](using ord: Ordering[A]): Ordering[List[A]] with

  def compare(xs: List[A], ys: List[A]) = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _)   => -1
    case (_, Nil)   => 1
    case (x :: xs1, y :: ys1) =>
      val c = ord.compare(x, y)
      if c != 0 then c else compare(xs1, ys1)
```

The given instance `listOrdering` takes type parameters and implicit parameters.

Conditional Instances

Given instances such as `listOrdering` that take implicit parameters are *conditional*:

- An ordering for lists with elements of type `T` exists only if there is an ordering for `T`.

This sort of conditional behavior is best implemented with type classes.

- Normal subtyping and inheritance cannot express this: a class either inherits a trait or doesn't.

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

```
def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
val xss: List[List[Int]] = ...

sort(xss)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

```
def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
val xss: List[List[Int]] = ...

sort[List[Int]](xss)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

```
def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
val xss: List[List[Int]] = ...

sort[List[Int]](xss)(using listOrdering)
```

Recursive Implicit Resolution

Given instances with implicit parameters are resolved recursively:

A given instance for the outer type is constructed first and then its implicit parameters are filled in in turn.

Example:

```
def sort[A](xs: List[A])(using Ordering[A]): List[A] = ...
val xss: List[List[Int]] = ...

sort[List[Int]](xss)(using listOrdering(using Ordering.Int))
```

Exercise

Implement an instance of the `Ordering` typeclass for pairs of type `(A, B)`, where `A`, `B` have `Ordering` instances defined on them.

Example use case: Consider a program for managing an address book. We would like to sort the addresses by zip codes first and then by street name. Two addresses with different zip codes are ordered according to their zip code, otherwise (when the zip codes are the same) the addresses are sorted by street name. E.g.

```
type Address = (Int, String)  // Zipcode, Street Name
val xs: List[Address] = ...
sort(xs)
```

Exercise

Implement an instance of the `Ordering` typeclass for pairs of type `(A, B)`, where `A`, `B` have `Ordering` instances defined on them.

```
given pairOrdering[A, B](using orda: Ordering[A], ordb: Ordering[B])
 : Ordering[(A, B)] with
  def compare(x: (A, B), y: (A, B)) =
    val c = orda.compare(x._1, y._1)
    if c != 0 then c else ordb.compare(x._2, y._2)
```

Type Classes and Extension Methods

Like any trait, a type class trait may define extension methods.

For, instance, the `Ordering` trait would usually contain comparison methods like this:

```
trait Ordering[A]:

  def compare(x: A, y: A): Int

  extension (x: A)
    def <  (y: A): Boolean = compare(x, y) < 0
    def <= (y: A): Boolean = compare(x, y) <= 0
    def >  (y: A): Boolean = compare(x, y) > 0
    def >= (y: A): Boolean = compare(x, y) >= 0
```

Visibility of Extension Methods

Extension methods on a type class trait are visible whenever a given instance for the trait is available.

For instance one can write:

```
def merge[T: Ordering](xs: List[T], ys: List[T]): Boolean = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

- There's no need to name and import the `Ordering` instance to get access to the extension method < on operands of type `T`.
- We have an `Ordering[T]` instance in scope, that's where the extension method comes from.

Summary

Type classes provide a way to turn types into values.

Unlike class extension, type classes

- can be defined at any time without changing existing code,
- can be conditional.

In Scala, type classes are constructed from parameterized traits and given instances.

Type classes give rise to a new kind of polymorphism, which is sometimes called *ad-hoc* polymorphism.

This means that the a type `TC[A]` has different implementations for different types `A`.

EPFL

Abstract Algebra and Type Classes

Principles of Functional Programming

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

We can then define methods that work for all semigroups. For instance:

```
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

Type Class Hierarchies

Algebraic type classes often form natural hierarchies. For instance, a *monoid* is defined as a semigroup with a left and right unit element.

Here's its natural definition:

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
```

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

```
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Using Context Bounds

In the previous example we had to pass an explicitly named type class instance `m: Monoid[T]` to `reduce`, so that we could refer to `m.unit`.

One could alternatively use a context bound and a `summon`.

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.reduceLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Streamlining Access

A simpler calling syntax can be obtained if we do some preparation in the `Monoid` trait itself.

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
object Monoid:
  def apply[T](using m: Monoid[T]): Monoid[T] = m
```

This defines a global function `Monoid.apply[T]` that returns the `Monoid[T]` instance that is currently visible.

With this helper, `reduce` can be written like this:

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.reduceLeft(Monoid[T].unit)(_.combine(_))
```

Multiple Typeclass Instances

It's possible to have several given instances for a typeclass/type pair. For instance, `Int` could be a `Monoid` in (at least) two ways:

- with `+` as `combine` and `0` as `unit`, or
- with `*` as `combine` and `1` as `unit`.

```
given sumMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x + y
  def unit: Int = 0

given prodMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x * y
  def unit: Int = 1
```

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

```
def sum(xs: List[Int]): Int = reduce(xs)(using sumMonoid)
def product(xs: List[Int]): Int = reduce(xs)(using prodMonoid)
```

What happens if you leave out the `using` arguments?

Exercise

Define the `sum` and `product` functions on `List[Int]` in terms of `reduce`.

```
def sum(xs: List[Int]): Int = reduce(xs)(using sumMonoid)
def product(xs: List[Int]): Int = reduce(xs)(using prodMonoid)
```

What happens if you leave out the `using` arguments?

An ambiguity error.

Typeclass Laws

Algebraic type classes are not just defined by their type signatures but also by the laws that hold for them.

For example, any given instance of `Monoid[T]` should satisfy the laws:

```
x.combine(y).combine(z)  ==  x.combine(y.combine(z))
        unit.combine(x)  ==  x
        x.combine(unit)  ==  x
```

where `x`, `y`, `z` are arbitrary values of type `T` and `unit = Monoid.unit[T]`.

The laws can be verified either by a formal or informal proof, or by testing them.

A good way to test that an instance is *lawful* is using randomized testing with a tool like ScalaCheck.

Context Passing

Principles of Functional Programming

Context Passing vs Type Classes

Type classes are about *type instances of generic traits*. E.g.:

- What is the definition of `TC[A]` for the type class trait `TC` and the type argument `A`?

If we want to make `A` a type parameter, we need an implicit parameter to go with it.

On the other hand, there are also uses for abstracting over values of a simple type, asking

- What is the currently valid definition of type `T`?

Example: Execution contexts

To do computations in parallel, runtimes need *thread schedulers*.

There's usually a default scheduler, but it should be possible to override that choice in parts of the code.

How are references to schedulers propagated?

In Scala, they are embedded in values of types `ExecutionContext`. The default is:

```
given global: ExecutionContext = ForkJoinContext()
```

This defines the execution context `global` as an alias of an existing value (i.e. a freshly created `ForkJoinContext`)

The evaluation of `ForkJoinContext` is done lazily: the `ForkJoinContext` is created the first time `global` is used.

Propagating Execution Contexts

Execution contexts rarely change, but they should be changeable everywhere.

This is a poster-child for implicit parameters.

```
def processItems(...)(using ExecutionContext) = ...
```

Other Use Cases

Passing a piece of the context as an implicit parameter of a certain type is quite common.

For instance, we might want to propagate implicitly

- the current configuration,
- the available set of capabilities,
- the security level in effect,
- the layout scheme to render some data,
- The persons that have access to some data.

Example: A Conference Management System

Let's say we design a system to discuss papers submitted to a conference.

- The papers have already been given a score by the reviewers.
- To discuss, reviewers need to see various pieces of information about the papers.
- Some reviewers are also authors of papers.
- An author of a paper should never see at this phase the score the paper received from the other reviewers.

Consequence: Every query of the conference needs to know who is seeing the results of the operation and this needs to be propagated.

For a given toplevel query the set of persons seeing its results will largely stay the same.

But it can change, for instance when a reviewer *delegates* part of the task to another person.

Outline

```
case class Person(name: String)
case class Paper(title: String, authors: List[Person], body: String)

object ConfManagement:
  type Viewers = Set[Person]

  class Conference(ratings: (Paper, Int)*):
    private val realScore = ratings.toMap

    def papers: List[Paper] = ratings.map(_._1).toList

    def score(paper: Paper, viewers: Viewers): Int =
      if paper.authors.exists(viewers.contains) then -100
      else realScore(paper)
```

Outline ctd

```
    def rankings(viewers: Viewers): List[Paper] =
      papers.sortBy(score(_, viewers)).reverse

    def ask[T](p: Person, query: Viewers => T) =
      query(Set(p))

    def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
      query(viewers + p)
  end Conference
end ConfManagement
```

- If one of the viewers is also an author if the paper, the score is *masked*, returning -100 instead of the real score.
- The same masking also has to be done in derived operations, such as `rankings`.

Example Dataset

```
val Smith  = Person(”Smith”)
val Peters = Person(”Peters”)
val Abel   = Person(”Abel”)
val Black  = Person(”Black”)
val Ed     = Person(”Ed”)

val conf = Conference(
  Paper(”How to grow beans”, List(Smith, Peters), ”...”) -> 92,
  Paper(”Organic gardening”, List(Abel, Peters), ”...”) -> 83,
  Paper(”Composting done right”, List(Black, Smith), ”...”) -> 99,
  Paper(”The secret life of snails”, authors = List(Ed), ”...”) -> 77
)
```

Example Query

Which authors have at least two papers with a score over 80?

```
def highlyRankedProlificAuthors(asking: Person): Set[Person] =
  def query(viewers: Viewers): Set[Person] =
    val highlyRanked =
      conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
    for
      p1 <- highlyRanked
      p2 <- highlyRanked
      author <- p1.authors
      if p1 != p2 && p2.authors.contains(author)
    yield author
  conf.ask(asking, query)
```

The answer depends on who is asking!

Tamper-Proofing

Problem: So far passing viewers is a *convention*.

Nothing prevents just passing the empty set of viewers to a query.

```
conf.rankings(Set()).takeWhile(conf.score(_, Set()) > 80)
```

Fix: Make the `Viewers` type alias *opaque*:

```
opaque type Viewers = Set[Person]
```

Opaque Type Aliases

Given an opaque type alias such as

```
object ConfManagement:
  opaque type Viewers = Set[Person]
```

the equality `Viewers = Set[Person]` is known only within the scope where the alias is defined. (in this case, within the `ConfManagement` object)

Everywhere else `Viewers` is treated as a separate, abstract type.

Why Does This Help Against Tampering?

When asking a query, we have to pass a `Viewers` set to the conference management methods.

But `Viewers` is an unknown abstract type; hence there is no way to create a `Viewers` instance outside the `ConfManagement` object.

So the only way to get a viewers value is in the parameter of a query, where the conference management system provides the actual value.

Therefore, in

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
```

we are *forced* to pass `viewers` on to `rankings` and `score` since that's the only `Viewers` value we have access to.

Caveat: This assumes that queries are not nested, since otherwise an inner query could access the `viewers` parameter of an outer one)

Discussion

Back to the conference management code:

- One downside is that we have to pass `viewers` arguments along everywhere they are needed.
- This seems pointless, since *by design* there is only a single value we could pass!
- It also quickly gets tedious as the codebase grows.
- Can't this be automated?

Discussion

Back to the conference management code:

- One downside is that we have to pass `viewers` arguments along everywhere they are needed.
- This seems pointless, since *by design* there is only a single value we could pass!
- It also quickly gets tedious as the codebase grows.
- Can't this be automated?

Of course: Just use implicit parameters.

Using using Clauses

```
class Conference(ratings: (Paper, Int)*):
  ...
  def score(paper: Paper)(viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)
  def rankings(viewers: Viewers): List[Paper] =
    papers.sortBy(score(_, viewers)).reverse
  def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
    query(viewers + p)
  ...
```

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
```

Using using Clauses

```
class Conference(ratings: (Paper, Int)*):
  ...
  def score(paper: Paper)(using viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)
  def rankings(using viewers: Viewers): List[Paper] =
    papers.sortBy(score(_)).reverse
  def delegateTo[T](p: Person, query: Viewers => T)(using viewers: Viewers):
    query(viewers + p)
  ...


conf.rankings.takeWhile(conf.score(_) > 80).toSet
```

Another Benefit of Opacity

The implicit parameters are of type `Viewers`, which is an opaque type alias.

This has another benefit: Since outside `ConfManagement`, `Viewers` is a type different from all others , there's no chance to connect `Viewers` implicit parameters with given instances for other types.

On the other hand, if `Viewers` was a regular type alias of `Set[Person]` we might accidentally have given instances for other sets of persons in scope, which would then be eligible candidates for `Viewers` parameters.

Be Specific

Morale: Given instances should have specific types and/or be local in scope.

For example, this is a terrible idea:

```
given Int = 1
def f(x: Int)(using delta: Int) = x + delta
```

Never use a common type such as `Int` or `String` as the type of a globally visible given instance!

Exercise

You have seen in week 4 an `enum` for arithmetic expressions. Let's augment it with a `Let` form:

```
enum Expr:
  case Number(num: Int)
  case Sum(x: Expr, y: Expr)
  case Prod(x: Expr, y: Expr)
  case Var(name: String)
  case Let(name: String, rhs: Expr, body: Expr)
import Expr._
```

Write an `eval` function for expressions of this type.

```
def eval(e: Expr): Int = ???
```

`Let("x", e1, e2)` should be evaluated like `{val x = e1; e2}`.
You can assume that every `Var(x)` occurs in the body `b` of an enclosing `Let(x, e, b)`.

Solution Hint

Use a map from variable names to their defined values as an implicit parameter.

The map is initially empty and is augmented in every `Let` node.

This suggests the following outline:

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = ???

  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            =>
    case Sum(x, y)            =>
    case Prod(x, y)           =>
    case Var(name)            =>
    case Let(name, rhs, body) =>
  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            => n
    case Sum(x, y)            =>
    case Prod(x, y)           =>
    case Var(name)            =>
    case Let(name, rhs, body) =>
  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            => n
    case Sum(x, y)            => recur(x) + recur(y)
    case Prod(x, y)           =>
    case Var(name)            =>
    case Let(name, rhs, body) =>
  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            => n
    case Sum(x, y)            => recur(x) + recur(y)
    case Prod(x, y)           => recur(x) * recur(y)
    case Var(name)            =>
    case Let(name, rhs, body) =>
  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            => n
    case Sum(x, y)            => recur(x) + recur(y)
    case Prod(x, y)           => recur(x) * recur(y)
    case Var(name)            => env(name)
    case Let(name, rhs, body) =>
  recur(e)(using Map())
```

Solution

```
def eval(e: Expr): Int =
  def recur(e: Expr)(using env: Map[String, Int]): Int = e match
    case Number(n)            => n
    case Sum(x, y)            => recur(x) + recur(y)
    case Prod(x, y)           => recur(x) * recur(y)
    case Var(name)            => env(name)
    case Let(name, rhs, body) => recur(body)(using env + (name -> recur(rhs)))
  recur(e)(using Map())
```

Implicit Function Types

Principles of Functional Programming

Repetitive Using Clauses

In last version of the conference management system of the last session we got rid of explicit `Viewers` arguments.

But we still need explicit `using` parameter clauses.

```
def score(paper: Paper)(using Viewers): Int = ...
def rankings(using Viewers): List[Paper] = ...
def delegateTo(p: Person, query: Viewers => T)(using Viewers): T = ...
```

Can we get rid of these as well?

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) =>
  papers.sortBy(score(_, viewers)).reverse
```

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The `?` signifies that we want the parameter `viewers` be implicit so that its arguments can be inferred.

What is its type?

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The `?` signifies that we want the parameter `viewers` be implicit so that its arguments can be inferred.

What is its type?

- For a normal anonymous function it would be:
  `Viewers => List[Paper]`
- For an anonymous functions with a `using` clause it is:
  `Viewers ?=> List[Paper]`

Implicit Function Types

`Viewers ?=> List[Paper]` is called an *implicit function type*.

There are two typing rules involving such types.

1. Implicit functions get their arguments inferred just like methods with using clauses. In

   ```
   val f: A ?=> B
   given a: A
   f
   ```

   the expression `f` expands to `f(using a)`.

Implicit Function Types

`Viewers ?=> List[Paper]` is called an *implicit function type*.

There are two typing rules involving such types.

1. Implicit functions get their arguments inferred just like methods with using clauses. In

   ```
   val f: A ?=> B
   given a: A
   f
   ```

   the expression `f` expands to `f(using a)`.
2. Implicit functions get created on demand.
   If the expected type of an expression `b` is `A ?=> B`, then `b` expands to the anonymous function `(_: A) ?=> b`.

Example Application

Let's use implicit function types in our conference management system.

First, introduce a type alias

```
type Viewed[T] = Viewers ?=> T
```

This is just for conciseness; `Viewed[T]` is easier to read than `Viewers ?=> T` and it expresses the point we want to make.

Example Application (2)

Now, perform the apply two changes:

1. Replace every method signature ending in

   ```
   (using Viewers): SomeType
   ```

   with

   ```
   : Viewed[SomeType]
   ```

Example Application (2)

Now, perform the apply two changes:

1. Replace every method signature ending in

   ```
   (using Viewers): SomeType
   ```

   with

   ```
   : Viewed[SomeType]
   ```

2. Replace function type parameter

   ```
   query: Viewers => SomeType
   ```

   with

   ```
   query: Viewed[SomeType]
   ```

Trade Types for Type Parameters

Implicit Parameters in using clauses trade *types* for *terms*:

- The developer writes down the required type of the parameter.
  The compiler infers an expression (i.e. a term) for it.

Trade Types for Type Parameters

Implicit Parameters in using clauses trade *types* for *terms*:

- The developer writes down the required type of the parameter.
  The compiler infers an expression (i.e. a term) for it.

Implicit Function Types go one step further. They trade types for parameters.

- The developer writes down the return type of the method.
  The compiler infers one or more method parameters that match the type.

Abstracting over Context Abstractions

Another way to look at it is to see implicit function types, as *second degree context abstractions*.

- Implicit parameters in `using` clauses abstract over the context at the call site.
  They are first-degree context abstractions.
- Implicit function types allow to abstract over using clauses (in the original sense: they allow to introduce a name such as `Viewed` that can be used instead of writing explicit parameter clauses).
- So, together with type aliases, they enable abstractions of context abstractions.