



# 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`.