```
# FIELDS:
#=======================================================================================#
"""
    abstract type Field end

Abstract type at the top of the type hierarchy for denoting fields over which
vector spaces
can be defined. Two common fields are `ℝ` and `ℂ`, representing the field of real
or complex
numbers respectively.
"""
abstract type Field end

struct RealNumbers <: Field end
struct ComplexNumbers <: Field end

const ℝ = RealNumbers()
const ℂ = ComplexNumbers()

Base.show(io::IO, ::RealNumbers) = print(io, "ℝ")
Base.show(io::IO, ::ComplexNumbers) = print(io, "ℂ")

Base.in(::Any, ::Field) = false
Base.in(::Real, ::RealNumbers) = true
Base.in(::Number, ::ComplexNumbers) = true

Base.@pure Base.issubset(::Type, ::Field) = false
Base.@pure Base.issubset(::Type{<:Real}, ::RealNumbers) = true
Base.@pure Base.issubset(::Type{<:Number}, ::ComplexNumbers) = true
Base.@pure Base.issubset(::RealNumbers, ::RealNumbers) = true
Base.@pure Base.issubset(::RealNumbers, ::ComplexNumbers) = true
Base.@pure Base.issubset(::ComplexNumbers, ::RealNumbers) = false
Base.@pure Base.issubset(::ComplexNumbers, ::ComplexNumbers) = true

# VECTOR SPACES:
#=======================================================================================#
"""
    abstract type VectorSpace end

Abstract type at the top of the type hierarchy for denoting vector spaces, or, more
accurately, 𝕜-linear categories.
"""
abstract type VectorSpace end

"""
    field(V::VectorSpace) -> Field

Return the field type over which a vector space is defined.
"""
function field end
field(V::VectorSpace) = field(typeof(V))

# Basic vector space methods
```

```
#-------------------------------
"""
    space(a) -> VectorSpace

Return the vector space associated to object `a`.
"""
function space end

"""
    dim(V::VectorSpace) -> Int

Return the total dimension of the vector space `V` as an Int.
"""
function dim end

"""
    dual(V::VectorSpace) -> VectorSpace

Return the dual space of `V`; also obtained via `V'`. It is assumed that
`typeof(V) == typeof(V')`.
"""
function dual end

"""
    isdual(V::ElementarySpace) -> Bool

Return wether an ElementarySpace `V` is normal or rather a dual space. Always
returns
`false` for spaces where `V == dual(V)`.
"""
function isdual end

# convenience definitions:
Base.adjoint(V::VectorSpace) = dual(V)
Base.:*(V1::VectorSpace, V2::VectorSpace) = ⊗(V1, V2)

# Hierarchy of elementary vector spaces
#---------------------------------------------
"""
    abstract type ElementarySpace{𝕜} <: VectorSpace end

Elementary finite-dimensional vector space over a field `𝕜` that can be used as
the index
space corresponding to the indices of a tensor.

Every elementary vector space should respond to the methods [`conj`](@ref) and
[`dual`](@ref), returning the complex conjugate space and the dual space
respectively. The
complex conjugate of the dual space is obtained as `dual(conj(V)) ===
conj(dual(V))`. These
different spaces should be of the same type, so that a tensor can be defined as an
element
of a homogeneous tensor product of these spaces.
"""
```

```julia
abstract type ElementarySpace{𝕜} <: VectorSpace end
const IndexSpace = ElementarySpace

field(::Type{<:ElementarySpace{𝕜}}) where {𝕜} = 𝕜

"""
    oneunit(V::S) where {S<:ElementarySpace} -> S

Return the corresponding vector space of type `S` that represents the trivial
one-dimensional space, i.e. the space that is isomorphic to the corresponding
field. Note
that this is different from `one(V::S)`, which returns the empty product space
`ProductSpace{S,0}(())`.
"""
Base.oneunit(V::ElementarySpace) = oneunit(typeof(V))


"""
    ⊕(V1::S, V2::S, V3::S...) where {S<:ElementarySpace} -> S

Return the corresponding vector space of type `S` that represents the direct sum
sum of the
spaces `V1`, `V2`, ... Note that all the individual spaces should have the same
value for
[`isdual`](@ref), as otherwise the direct sum is not defined.
"""
function ⊕ end
⊕(V1, V2, V3, V4...) = ⊕(⊕(V1, V2), V3, V4...)


"""
    ⊗(V1::S, V2::S, V3::S...) where {S<:ElementarySpace} -> S

Create a [`ProductSpace{S}(V1, V2, V3...)`](@ref) representing the tensor product
of several
elementary vector spaces. For convience, Julia's regular multiplication operator
`*` applied
to vector spaces has the same effect.

The tensor product structure is preserved, see [`fuse`](@ref) for returning a
single
elementary space of type `S` that is isomorphic to this tensor product.
"""
function ⊗ end
⊗(V1, V2, V3, V4...) = ⊗(⊗(V1, V2), V3, V4...)

"""
    fuse(V1::S, V2::S, V3::S...) where {S<:ElementarySpace} -> S
    fuse(P::ProductSpace{S}) where {S<:ElementarySpace} -> S

Return a single vector space of type `S` that is isomorphic to the fusion product
of the
individual spaces `V1`, `V2`, ..., or the spaces contained in `P`.
"""
```

```julia
function fuse end
fuse(V::ElementarySpace) = V
fuse(V1::VectorSpace, V2::VectorSpace, V3::VectorSpace...) =
    fuse(fuse(fuse(V1), fuse(V2)), V3...)
    # calling fuse on V1 and V2 will allow these to be `ProductSpace`

"""
    flip(V::S) where {S<:ElementarySpace} -> S

Return a single vector space of type `S` that has the same value of
[`isdual`](@ref) as
`dual(V)`, but yet is isomorphic to `V` rather than to `dual(V)`. The spaces
`flip(V)` and
`dual(V)` only differ in the case of [`RepresentationSpace{G}`](@ref).
"""
function flip end

"""
    conj(V::S) where {S<:ElementarySpace} -> S

Return the conjugate space of `V`.

For `field(V)==ℝ`, `conj(V) == V`. It is assumed that `typeof(V) ==
typeof(conj(V))`.
"""
Base.conj(V::ElementarySpace{ℝ}) = V

"""
    abstract type InnerProductSpace{𝕜} <: ElementarySpace{𝕜} end

Abstract type for denoting vector with an inner product and a corresponding
metric, which
can be used to raise or lower indices of tensors.
"""
abstract type InnerProductSpace{𝕜} <: ElementarySpace{𝕜} end

"""
    abstract type EuclideanSpace{𝕜} <: InnerProductSpace{𝕜} end

Abstract type for denoting real or complex spaces with a standard (Euclidean)
inner product
(i.e. orthonormal basis), such that the dual space is naturally isomorphic to the
conjugate
space (in the complex case) or even to the space itself (in the real case), also
known as
the category of finite-dimensional Hilbert spaces ``FdHilb``.
"""
abstract type EuclideanSpace{𝕜} <: InnerProductSpace{𝕜} end # 𝕜 should be ℝ or ℂ

dual(V::EuclideanSpace) = conj(V)
isdual(V::EuclideanSpace{ℝ}) = false
# dual space is naturally isomorphic to conjugate space for inner product spaces
```

```julia
# representation spaces: we restrict to complex Euclidean space supporting unitary
  representations
"""
    abstract type RepresentationSpace{G<:Sector} <: EuclideanSpace{ℂ} end

Complex Euclidean space with a direct sum structure corresponding to different
superselection sectors of type `G<:Sector`, e.g. the elements or irreps of a
compact or
finite group, or the labels of a unitary fusion category.
"""
abstract type RepresentationSpace{G<:Sector} <: EuclideanSpace{ℂ} end
const Rep{G<:Sector} = RepresentationSpace{G}

"""
    sectortype(a) -> Type{<:Sector}

Return the type of sector over which object `a` (e.g. a representation space or a
tensor) is
defined. Also works in type domain.
"""
sectortype(V::VectorSpace) = sectortype(typeof(V))
sectortype(::Type{<:ElementarySpace}) = Trivial
sectortype(::Type{<:RepresentationSpace{G}}) where {G} = G

"""
    hassector(V::VectorSpace, a::Sector) -> Bool

Return whether a vector space `V` has a subspace corresponding to sector `a` with
non-zero dimension, i.e. `dim(V, a) > 0`.
"""
hassector(V::ElementarySpace, ::Trivial) = dim(V) != 0
Base.axes(V::ElementarySpace, ::Trivial) = axes(V)

struct TrivialOrEmptyIterator
    isempty::Bool
end
Base.IteratorSize(::TrivialOrEmptyIterator) = Base.HasLength()
Base.IteratorEltype(::TrivialOrEmptyIterator) = Base.HasEltype()
Base.isempty(V::TrivialOrEmptyIterator) = V.isempty
Base.length(V::TrivialOrEmptyIterator) = isempty(V) ? 0 : 1
Base.eltype(::TrivialOrEmptyIterator) = Trivial
function Base.iterate(V::TrivialOrEmptyIterator, state = true)
    return isempty(V) == state ? nothing : (Trivial(), false)
end

"""
    sectors(V::ElementarySpace)

Return an iterator over the different sectors of `V`.
"""
sectors(V::ElementarySpace) = TrivialOrEmptyIterator(dim(V) == 0)
dim(V::ElementarySpace, ::Trivial) =
    sectortype(V) == Trivial ? dim(V) : throw(SectorMismatch())
```

```julia
# Composite vector spaces
#--------------------------
"""
    abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end

Abstract type for composite spaces that are defined in terms of a number of
elementary
vector spaces of a homogeneous type `S<:ElementarySpace{𝕜}`.
"""
abstract type CompositeSpace{S<:ElementarySpace} <: VectorSpace end

spacetype(S::Type{<:ElementarySpace}) = S
spacetype(V::ElementarySpace) = typeof(V) # = spacetype(typeof(V))
spacetype(::Type{<:CompositeSpace{S}}) where S = S
spacetype(V::CompositeSpace) = spacetype(typeof(V)) # = spacetype(typeof(V))

field(P::Type{<:CompositeSpace}) = field(spacetype(P))
sectortype(P::Type{<:CompositeSpace}) = sectortype(spacetype(P))

# make ElementarySpace instances behave similar to ProductSpace instances
blocksectors(V::ElementarySpace) = sectors(V)
blockdim(V::ElementarySpace, c::Sector) = dim(V, c)

# Specific realizations of ElementarySpace types
#------------------------------------------------
# spaces without internal structure
include("cartesianspace.jl")
include("complexspace.jl")
include("generalspace.jl")
include("representationspace.jl")

# Specific realizations of CompositeSpace types
#-----------------------------------------------
include("productspace.jl")

# Other examples might include:
# braidedspace and fermionspace
# symmetric and antisymmetric subspace of a tensor product of identical vector
  spaces
# ...

# HomSpace: space of morphisms
#------------------------------
include("homspace.jl")

# Partial order for vector spaces
#---------------------------------
"""
    isisomorphic(V1::VectorSpace, V2::VectorSpace)
    V1 ≅ V2

Return if `V1` and `V2` are isomorphic, meaning that there exists isomorphisms
from `V1` to
`V2`, i.e. morphisms with left and right inverses.
```

```julia
"""
function isisomorphic(V1::VectorSpace, V2::VectorSpace)
    spacetype(V1) == spacetype(V2) || return false
    for c in union(blocksectors(V1), blocksectors(V2))
        if blockdim(V1, c) != blockdim(V2, c)
            return false
        end
    end
    return true
end

"""
    ismonomorphic(V1::VectorSpace, V2::VectorSpace)
    V1 ≾ V2

Return whether there exist monomorphisms from `V1` to `V2`, i.e. 'injective'
morphisms with
left inverses.
"""
function ismonomorphic(V1::VectorSpace, V2::VectorSpace)
    spacetype(V1) == spacetype(V2) || return false
    for c in blocksectors(V1)
        if blockdim(V1, c) > blockdim(V2, c)
            return false
        end
    end
    return true
end

"""
    isepimorphic(V1::VectorSpace, V2::VectorSpace)
    V1 ≿ V2

Return whether there exist epimorphisms from `V1` to `V2`, i.e. 'surjective'
morphisms with
right inverses.
"""
function isepimorphic(V1::VectorSpace, V2::VectorSpace)
    spacetype(V1) == spacetype(V2) || return false
    for c in blocksectors(V1)
        if blockdim(V1, c) < blockdim(V2, c)
            return false
        end
    end
    return true
end

# unicode alternatives
const ≅ = isisomorphic
const ≾ = ismonomorphic
const ≿ = isepimorphic

<(V1::VectorSpace, V2::VectorSpace) = V1 ≾ V2 && !(V1 ≿ V2)
>(V1::VectorSpace, V2::VectorSpace) = V1 ≿ V2 && !(V1 ≾ V2)
```

```
"""
    infinum(V1::ElementarySpace, V2::ElementarySpace, V3::ElementarySpace...)

Return the infinum of a number of elementary spaces, i.e. an instance
`V::ElementarySpace`
such that `V ≾ V1`, `V ≾ V2`, ... and no other `W ≻ V` has this property. This
requires
that all arguments have the same value of `isdual( )`, and also the return value
`V` will
have the same value.
"""
infinum(V1::ElementarySpace, V2::ElementarySpace, V3::ElementarySpace...) =
    infinum(infinum(V1, V2), V3...)

"""
    supremum(V1::ElementarySpace, V2::ElementarySpace, V3::ElementarySpace...)

Return the supremum of a number of elementary spaces, i.e. an instance
`V::ElementarySpace`
such that `V ≿ V1`, `V ≿ V2`, ... and no other `W ≺ V` has this property. This
requires
that all arguments have the same value of `isdual( )`, and also the return value
`V` will
have the same value.
"""
supremum(V1::ElementarySpace, V2::ElementarySpace, V3::ElementarySpace...) =
    supremum(supremum(V1, V2), V3...)

import Base: min, max
Base.@deprecate min(V1::ElementarySpace, V2::ElementarySpace) infinum(V1,V2)
Base.@deprecate max(V1::ElementarySpace, V2::ElementarySpace) supremum(V1,V2)
```