```julia
"""
    struct ProductSpace{S<:ElementarySpace, N} <: CompositeSpace{S}

A `ProductSpace` is a tensor product space of `N` vector spaces of type
`S<:ElementarySpace`. Only tensor products between [`ElementarySpace`](@ref)
objects of the
same type are allowed.
"""
struct ProductSpace{S<:ElementarySpace, N} <: CompositeSpace{S}
    spaces::NTuple{N, S}
end
ProductSpace(spaces::Vararg{S,N}) where {S<:ElementarySpace, N} =
    ProductSpace{S,N}(spaces)
ProductSpace{S,N}(spaces::Vararg{S,N}) where {S<:ElementarySpace, N} =
    ProductSpace{S,N}(spaces)
ProductSpace{S}(spaces) where {S<:ElementarySpace} =
ProductSpace{S,length(spaces)}(spaces)
ProductSpace(P::ProductSpace) = P

# Corresponding methods
#-----------------------
"""
    dims(::ProductSpace{S,N}) -> Dims{N} = NTuple{N,Int}

Return the dimensions of the spaces in the tensor product space as a tuple of
integers.
"""
dims(P::ProductSpace) = map(dim, P.spaces)
dim(P::ProductSpace, n::Int) = dim(P.spaces[n])
dim(P::ProductSpace) = prod(dims(P))

Base.axes(P::ProductSpace) = map(axes, P.spaces)
Base.axes(P::ProductSpace, n::Int) = axes(P.spaces[n])

dual(P::ProductSpace{<:ElementarySpace,0}) = P
dual(P::ProductSpace) = ProductSpace(map(dual, reverse(P.spaces)))

# Base.conj(P::ProductSpace) = ProductSpace(map(conj, P.spaces))

function Base.show(io::IO, P::ProductSpace{S}) where {S<:ElementarySpace}
    spaces = P.spaces
    if length(spaces) == 0
        print(io,"ProductSpace{", S, ",0}")
    end
    if length(spaces) == 1
        print(io,"ProductSpace")
    end
    print(io,"(")
    for i in 1:length(spaces)
        i==1 || print(io," ⊗ ")
        show(io, spaces[i])
    end
    print(io,")")
end
```

```
# more specific methods
"""
    sectors(P::ProductSpace{S,N}) where {S<:ElementarySpace}

Return an iterator over all possible combinations of sectors (represented as an
`NTuple{N,sectortype(S)}`) that can appear within the tensor product space `P`.
"""
sectors(P::ProductSpace) = _sectors(P, sectortype(P))
_sectors(P::ProductSpace{<:ElementarySpace, N}, ::Type{Trivial}) where {N} =
    (ntuple(n->Trivial(), StaticLength{N}()),) # speed up sectors for ungraded
        spaces
_sectors(P::ProductSpace{<:ElementarySpace, N}, ::Type{<:Sector}) where {N} =
    product(map(sectors, P.spaces)...)


"""
    hassector(P::ProductSpace{S,N}, s::NTuple{N,sectortype(S)}) where
{S<:ElementarySpace}
    -> Bool

Query whether `P` has a non-zero degeneracy of sector `s`, representing a
combination of
sectors on the individual tensor indices.
"""
hassector(V::ProductSpace{<:ElementarySpace,N}, s::NTuple{N}) where {N} =
    reduce(&, map(hassector, V.spaces, s); init = true)

"""
    dims(P::ProductSpace{S,N}, s::NTuple{N,sectortype(S)}) where
{S<:ElementarySpace}
    -> Dims{N} = NTuple{N,Int}

Return the degeneracy dimensions corresponding to a tuple of sectors `s` for each
of the
spaces in the tensor product `P`.
"""
dims(P::ProductSpace{<:ElementarySpace, N}, sector::NTuple{N,<:Sector}) where {N} =
    map(dim, P.spaces, sector)

"""
    dims(P::ProductSpace{S,N}, s::NTuple{N,sectortype(S)}) where
{S<:ElementarySpace}
    -> Int

Return the total degeneracy dimension corresponding to a tuple of sectors for each
of the
spaces in the tensor product, obtained as `prod(dims(P, s))``.
"""
dim(P::ProductSpace{<:ElementarySpace, N}, sector::NTuple{N,<:Sector}) where {N} =
    reduce(*, dims(P, sector); init = 1)

Base.axes(P::ProductSpace{<:ElementarySpace,N}, sectors::NTuple{N,<:Sector}) where
{N} =
```

```julia
    map(axes, P.spaces, sectors)

"""
    blocksectors(P::ProductSpace)

Return an iterator over the different unique coupled sector labels, i.e. the different
fusion outputs that can be obtained by fusing the sectors present in the different spaces
that make up the `ProductSpace` instance.
"""
function blocksectors(P::ProductSpace{S,N}) where {S,N}
    G = sectortype(S)
    if G == Trivial
        return TrivialOrEmptyIterator(dim(P) == 0)
    end
    bs = Vector{G}()
    if N == 0
        push!(bs, one(G))
    elseif N == 1
        for s in sectors(P)
            push!(bs, first(s))
        end
    else
        for s in sectors(P)
            for c in ⊗(s...)
                if !(c in bs)
                    push!(bs, c)
                end
            end
        end
        # return foldl(union!, Set{G}(), (⊗(s...) for s in sectors(P)))
    end
    return bs
end

"""
    blockdim(P::ProductSpace, c::Sector)

Return the total dimension of a coupled sector `c` in the product space, by summing over
all `dim(P, s)` for all tuples of sectors `s::NTuple{N,<:Sector}` that can fuse to `c`,
counted with the correct multiplicity (i.e. number of ways in which `s` can fuse to `c`).
"""
function blockdim(P::ProductSpace, c::Sector)
    sectortype(P) == typeof(c) || throw(SectorMismatch())
    d = 0
    for s in sectors(P)
        ds = dim(P, s)
        d += length(fusiontrees(s, c))*ds
    end
    return d
```

```
end

Base.:(==)(P1::ProductSpace, P2::ProductSpace) = (P1.spaces == P2.spaces)

Base.hash(P::ProductSpace, h::UInt) = hash(P.spaces, h)

# Default construction from product of spaces
#-----------------------------------------------
⊗(V1::S, V2::S) where {S<:ElementarySpace}= ProductSpace((V1, V2))
⊗(P1::ProductSpace{S}, V2::S) where {S<:ElementarySpace} =
    ProductSpace(tuple(P1.spaces..., V2))
⊗(V1::S, P2::ProductSpace{S}) where {S<:ElementarySpace} =
    ProductSpace(tuple(V1, P2.spaces...))
⊗(P1::ProductSpace{S}, P2::ProductSpace{S}) where {S<:ElementarySpace} =
    ProductSpace(tuple(P1.spaces..., P2.spaces...))
⊗(P::ProductSpace{S,0}, ::ProductSpace{S,0}) where {S<:ElementarySpace} = P
⊗(P::ProductSpace{S}, ::ProductSpace{S,0}) where {S<:ElementarySpace} = P
⊗(::ProductSpace{S,0}, P::ProductSpace{S}) where {S<:ElementarySpace} = P
⊗(V::ElementarySpace) = ProductSpace((V,))
⊗(P::ProductSpace) = P

# unit element with respect to the monoidal structure of taking tensor products
"""
    one(::S) where {S<:ElementarySpace} -> ProductSpace{S,0}
    one(::ProductSpace{S}) where {S<:ElementarySpace} -> ProductSpace{S,0}

Return a tensor product of zero spaces of type `S`, i.e. this is the unit object
under the
tensor product operation, such that `V ⊗ one(V) == V`.
"""
Base.one(V::VectorSpace) = one(typeof(V))
Base.one(::Type{<:ProductSpace{S}}) where {S<:ElementarySpace} =
ProductSpace{S,0}(())
Base.one(::Type{S}) where {S<:ElementarySpace} = ProductSpace{S,0}(())

Base.convert(::Type{<:ProductSpace}, V::ElementarySpace) = ProductSpace((V,))
Base.:^(V::ElementarySpace, N::Int) = ProductSpace{typeof(V), N}(ntuple(n->V, N))
Base.:^(V::ProductSpace, N::Int) = ⊗(ntuple(n->V, N)...)
Base.literal_pow(::typeof(^), V::ElementarySpace, p::Val{N}) where N =
    ProductSpace{typeof(V), N}(ntuple(n->V, p))
Base.convert(::Type{S}, P::ProductSpace{S,0}) where {S<:ElementarySpace} =
oneunit(S)
Base.convert(::Type{S}, P::ProductSpace{S}) where {S<:ElementarySpace} =
fuse(P.spaces...)
fuse(P::ProductSpace{S,0}) where {S<:ElementarySpace} = oneunit(S)
fuse(P::ProductSpace{S}) where {S<:ElementarySpace} = fuse(P.spaces...)

# Functionality for extracting and iterating over spaces
#-----------------------------------------------------------
Base.length(P::ProductSpace) = length(P.spaces)
Base.getindex(P::ProductSpace, n::Integer) = P.spaces[n]
# Base.getindex(P::ProductSpace{S}, I::NTuple{N,Integer}) where
  {S<:ElementarySpace,N} =
#     ProductSpace{S,N}(TupleTools.getindices(P.spaces, I))
```

```julia
Base.iterate(P::ProductSpace, args...) = Base.iterate(P.spaces, args...)
# Base.indexed_iterate(P::ProductSpace, args...) = Base.indexed_iterate(P.spaces,
  args...)

Base.eltype(::Type{<:ProductSpace{S}}) where {S<:ElementarySpace} = S
Base.eltype(P::ProductSpace) = eltype(typeof(P))

Base.IteratorEltype(::Type{<:ProductSpace}) = Base.HasEltype()
Base.IteratorSize(::Type{<:ProductSpace}) = Base.HasLength()
```