# Model object concept

Physical models are implemented in model objects. That means the code lines/equations compute results based on input values.

*Example 1. Model object*

> A heating model takes room air temperature (state dependency) and a scheduled setpoint (time dependency) and computes a heating load as result.

The result is stored in a **persistent memory location** where dependent models can directly access this value.

## Model instances

There can be several model instances - the actual object code resides only once in solver memory, but the functions are executed several times for each individual object (=model instance).

*Example 2. Model instance*

> You may have a model that computes heat flux between walls and zones. For each wall-zone interface, an object is instantiated.

Each model instance stores its result values in own memory.

## Model results

### Publishing model results

The models must tell the solver framework what kind of outputs they generate. For this purpose, the framework requests a reference type (prefix) via `AbstractModel::referenceType()`.

*Example 3. Reference types*

> `MRT_ZONE` or `MRT_SCHEDULE`

Also, each invididual result variable is published by the model as `QuantityDescription`. The framework requests these via call to `AbstractModel::resultDescriptions()`. Here, the model will store for each computed quantity the following information:

- id-name (e.g. "Temperature")
- physical display unit (e.g. "C") - interpreted as "display unit", calculations are always done in base SI units
- a descriptive text (e.g. "Room air temperature")
- physical limits (min/max values)
- flag indicating whether this value will be constant over the lifetime of the solver

**NOTE**

A note on units. The results are stored *always* in base SI units according to the UnitList. A well-behaving model will always store the result value in the basic SI unit, that is "K" for temperatures, "W" for thermal loads and so on (see `IBK::UnitList`).

The unit is really only provided for error message outputs and for checking of the base SI unit matches the base SI unit of the requested unit (as an additional sanity check).

## Vector-valued results

Sometimes, a model may compute a vector of values.

*Example 4. Vector results*

A ventilation model may compute ventilation heat loads for a number of zones (zones are identified via object lists).

When a so-called vector-valued quantity is generated, the following additional information is provided:

- whether index-based or id-based access is anticipated
- a list of ids/indexes matching the individual positions in the vector (the size of the vector is also the size of the vector-valued quantities)

*Example 5. ID-access example*

The aforementioned ventilation model may be published for zones with IDs 1,4,5,10 and 11. Then the resulting ventilation heat losses will be defined for those zones only. Hence, the published quantity will look like:

```
- name = "NaturalVentilationHeadLoad"
- unit = "W"
- description = "Natural ventilation heat load"
- index-key-type = "ID"
- ID-list = {1,4,5,10,11}
```

**IMPORTANT**

The id-names of quantities are unique within a model object, and vector-valued quantities may not have the same ID name as scalar quantities. More precisely, variables need to be globally unique (see lookup rules below). This means that when there are two models with the same model-reference-type, the variable names must be unique among all models with the same reference-type.

For example: from a zone parametrization several models are instantiated with the same reference-type `MRT_ZONE`, for example `RoomBalanceModel` and `RoomStatesModel`. The resulting variables in both models must not have the same ID name, in order to be uniquely identified on global scope.

## Initializing memory for result values

Each model object is requested by the framework in a call to `AbstractModel::initResults()` to reserve memory for its

result variables. For scalar variables this is usually just resizing of a vector of doubles to the required number of result values. For vector-valued quantities the actual size may only be known later, so frequently here just the vectors are created and their actual size is determined later.

Since the information collected in `initResults()` is also needed when publishing the results to the framework, the function `AbstractModel::initResults()` is called before `AbstractModel::resultDescriptions()`.

## Resolving persistant pointers to result locations

Scalar variables are stored in double variables of the model. When using the convenience implementation in `DefaultModel` these are stored in vector `m_results`.

Vector-valued variables are stored in consecutive memory arrays with size matching the size of the vector. When using the `DefaultModel` implementation, these are stored in `m_vectorValuedResults`, which is a vector of `VectorValuedResults`.

Later, when the framework connects model inputs with results, the framework requests models to provide persistant memory locations for previously published results. This is done by calling `AbstractModel::resultValueRef()` with a a quantity identification object `QuantityName`.

In order to uniquely identify a result variable **within** a model, two things are needed:

- the ID name of the variable,
- and, *only in the case of vector-valued quantities*, the index/id.

| | |
|---|---|
| **NOTE** | To identify an element within a vector-valued result it is not necessary to specify whether it should be index or id based - the model publishing the result defines whether it will be index or id based access. |

Naturally, for scalar result variables the index/id property is ignored.

The `QuantityName` struct contains this information (a string and an integer value).

Now the model searches through its own results and tries to find a matching variable. In case of vector-valued quantities it also checks if the requested id is actually present in the model, and in case of index-based access, a check is done if the index is in the allowed range (0…size-1).

If a quantity could be found, the corresponding memory address is returned, otherwise a nullptr. The framework now can take the address and pass it to any object that requires this input.

## Global lookup rules/global variable referencing

To uniquely reference a resulting variable (and its persistent memory location), first the actual model object/instance need to be selected with the following properties:

- the type of object to search for (= reference-type property), for example `MRT_ZONE` or `MRT_CONSTRUCTIONINSTANCE`
- ID of the referenced object, i.e. zone id oder construction id.

Some model objects exist only once, for example schedules or climatic loads. Here, the reference-type is already enough to uniquely select the object.

Usually, the information above does select several objects that have been created from the parametrization related to that ID. For example, the zone parameter block for some zone ID generates several zone-related model instances, all of which have the same ID. Since their result variables are all different, the framework simply searches through all those objects until the correct variable is found. These model implementations can be thought of as one model whose equations are split up into several implementation units.

The actual variable *within* the selected object is found by ID name and optionally vector-element id/index, as described above.

The data is collected in the class `InputReference`:

- `ObjectReference` ( holds reference-type and referenced-object ID )
- `QuantityName` (holds variable name and in case of vector-valued quantities also ID/index)

Also, it is possible to specify a `constant` flag to indicate, that during iterations over cycles the variable is to be treated constant.

| | |
|---|---|
| **IMPORTANT** | If several model objects are addressed by the same reference-type and ID (see example with models from zone parameter block), the variable names must be unique among all of these models. |

# Model inputs

Similarly as output variables, model objects need input variables. They publish these variables similarly to results when the framework requests them in a call to `AbstractStateDependency::inputReferences()`. The information on requested results is stored in objects of type `InputReference` again.

| | |
|---|---|
| **NOTE** | A model may request scalar variable inputs only, even if the providing model generates these as a vector-valued quantities. |

*Example 6. Input reference to a zone air temperature inside a ventilation model*

Then the inputs required by the ventilation model can be formulated with the follwing data:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "Temperature"
```

Given that information, the framework can effectively look-up the required variables and tell the object by calling `AbstractStateDependency::setInputValueRef()` which variable identified through `QuantityName` has which memory address.

## FMI Input variable overrides

Any input variable requested by any other model can be overridden by FMU import variables. When the framework looks up global model references, and an FMU import model is present/parametrized, then first the FMI generated quantity descriptions are checked. The FMU import variables are exported as global variable references (with ObjectReference). Since these are then the same global variable identifiers as published by the models, they are

found first in the search and dependent models will simply store points to the FMU variable memory.

## FMI Export variable input references

When FMI Export is parametrized, a list of global variable IDs to be exported is defined. For each of these variables an input reference is generated and the FMU export model gets addresses to these memory locations.

## Examples for referenced input quantities

### Setpoint from schedules

Schedules are defined for object lists. Suppose you have an object list name "Living rooms" and corresponding heating/cooling setpoints.

Now a heating model may be defined that computes heating loads for a given zone. The heating model is implemented with a simple P-controller, that requires zone air temperature and zone heating setpoint.

Definition of the input reference for the zone air temperature is done as in the example above. The setpoint will be similarly referenced:

```
- reference type = MRT_ZONE
- object_id = 15 (id of the zone)
- name = "HeatingSetPoint"
```

Now the models, that are created from the zone-parameter block