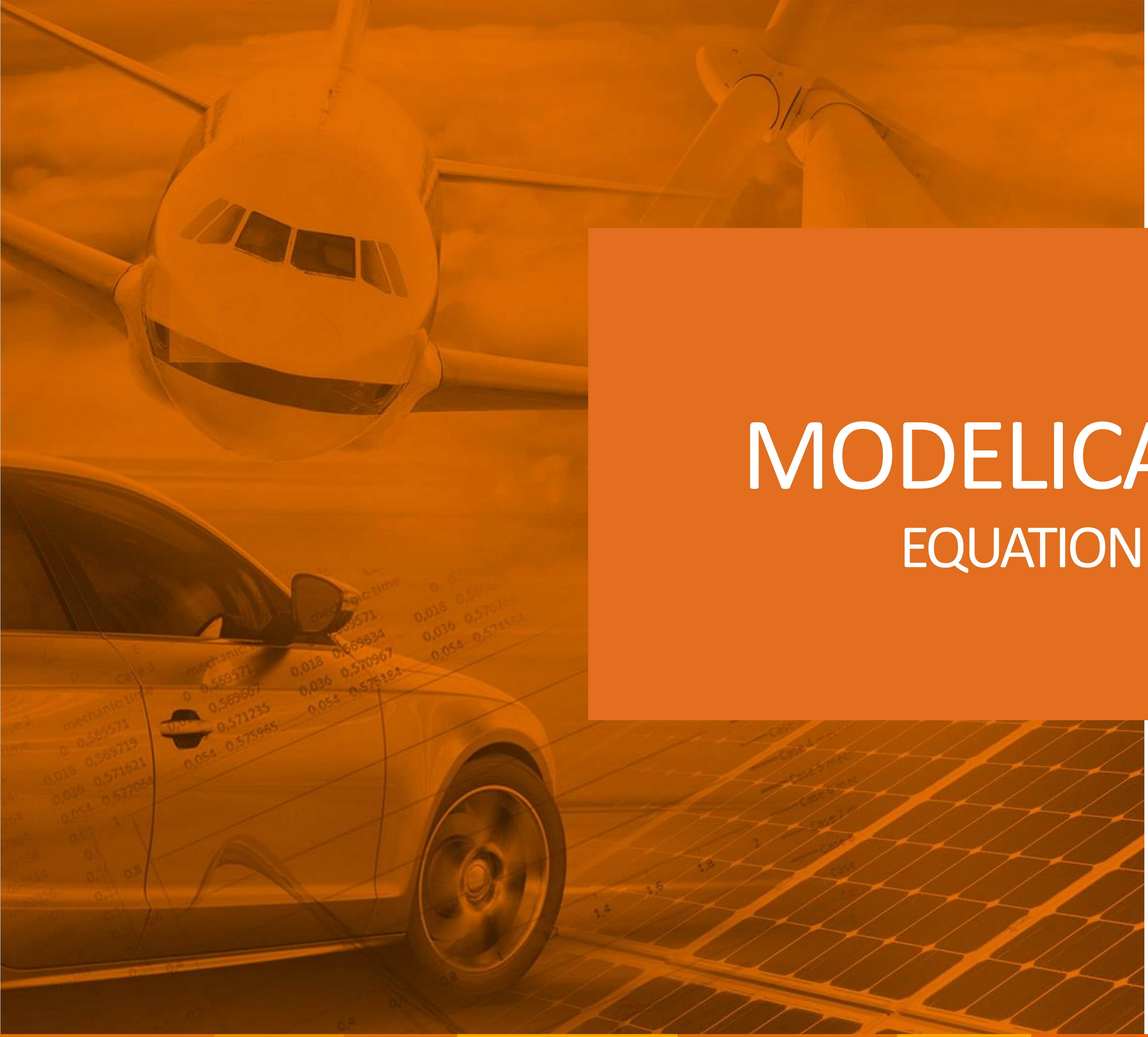


MODELICA LANGUAGE

EQUATION BASED COMPONENTS

Lecture 3.1

Modelon

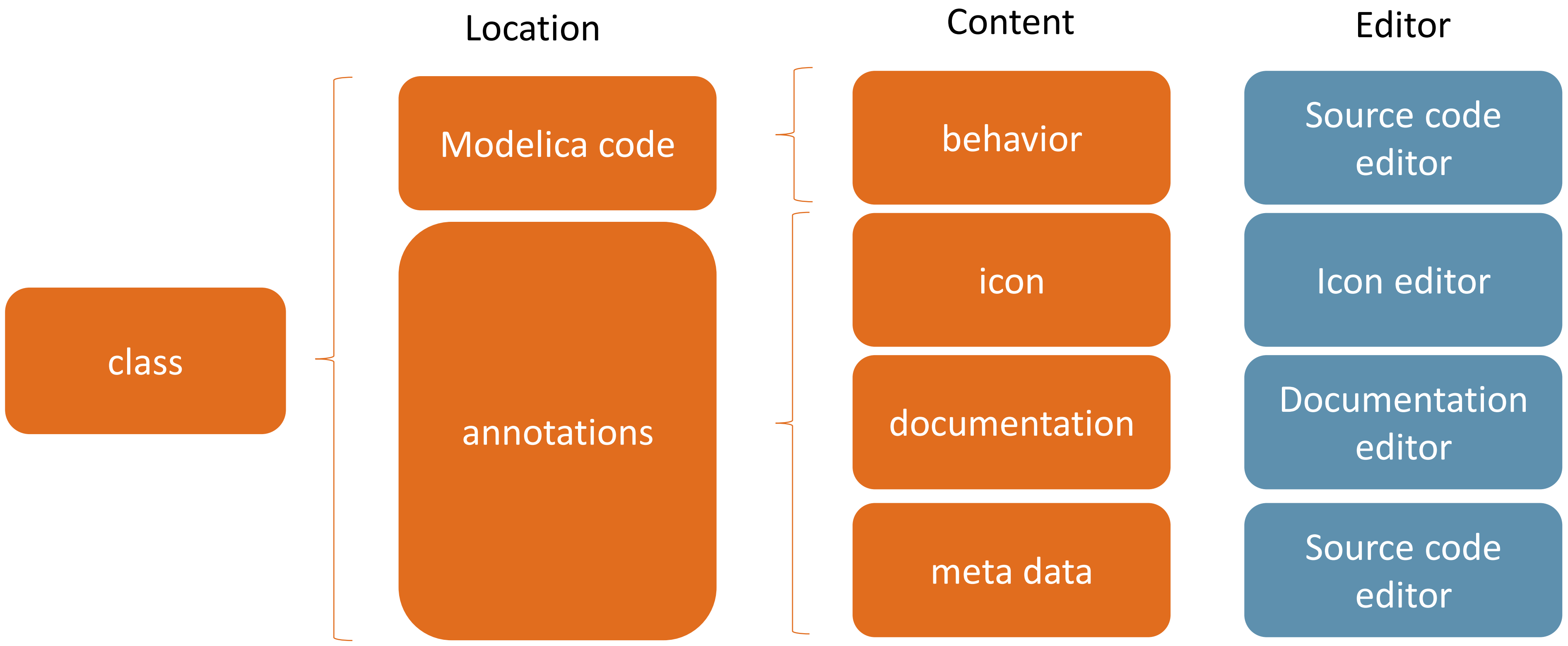
OVERVIEW

- Modelica class container
- Accessing the source code
- Modelica at a glance
- Variables and types
- Arrays and matrices
- Equation and algorithm
- Operators and statements
- Connectors and connect()
- Balancing concept and partial
- Inheritance v/s Instantiation

MODELICA CLASS CONTAINER

CLASSES – INFORMATION CONTENT

- Modelica classes are containers with information, defined by the Modelica Language Specification



Location

Content

Editor

class

Modelica code

annotations

behavior

icon

documentation

meta data

Source code editor

Icon editor

Documentation editor

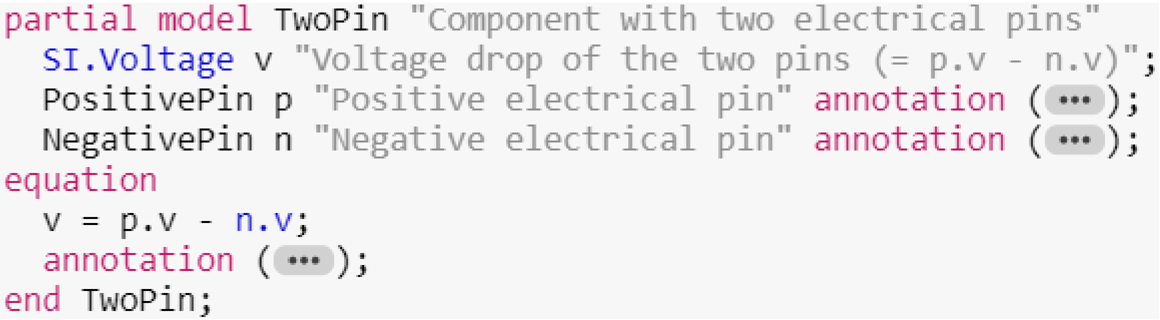
Source code editor

DIFFERENT KEYWORDS FOR CLASSES

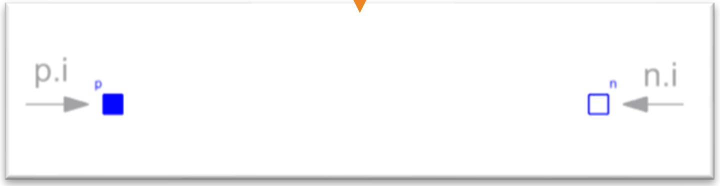
- **class**: any object is a class – used only when unspecific, e.g. documentation classes
- **package**: container for more classes – used to structure a set of models or a set of properties (e.g. fluid properties). Can only contain classes and constants
- **connector**: defines interfaces of models
- **model**: main class for physical behavior representation, using equations
- **block**: class for block diagrams – require input/output connectors
- **function**: to implement algorithm that relate output to input variables
- **record**: container for variables of any variability
- **type**: “refined class” so that its instance would be more specific – e.g. enumeration.
- (operator: mostly used for building blocks of your model – e.g. Complex())

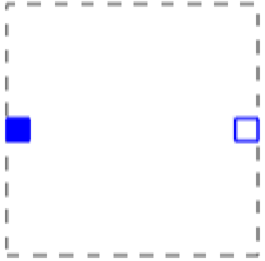
WHAT A MODEL SHOULD CONTAIN

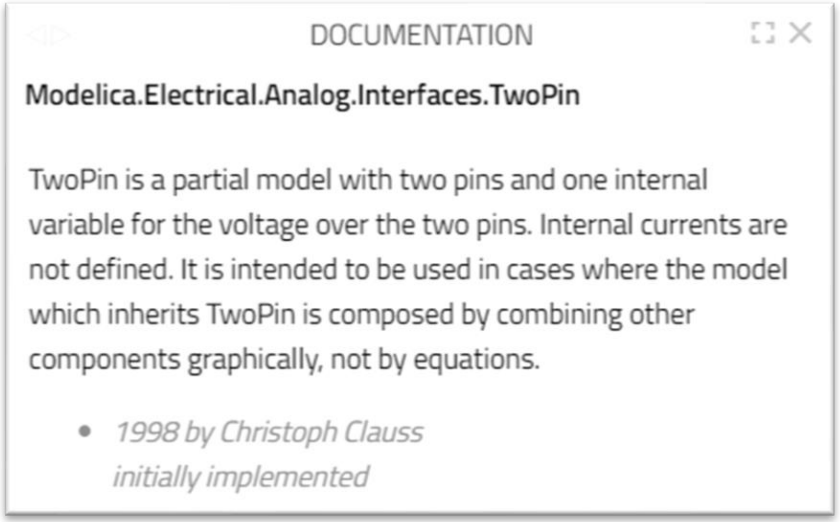
1. Modelica code
2. Documentation
3. Icon
4. Graphical representation



```
partial model TwoPin "Component with two electrical pins"
  SI.Voltage v "Voltage drop of the two pins (= p.v - n.v)";
  PositivePin p "Positive electrical pin" annotation (•••);
  NegativePin n "Negative electrical pin" annotation (•••);
equation
  v = p.v - n.v;
  annotation (•••);
end TwoPin;
```







Modelica.Electrical.Analog.Interfaces.TwoPin

TwoPin is a partial model with two pins and one internal variable for the voltage over the two pins. Internal currents are not defined. It is intended to be used in cases where the model which inherits TwoPin is composed by combining other components graphically, not by equations.

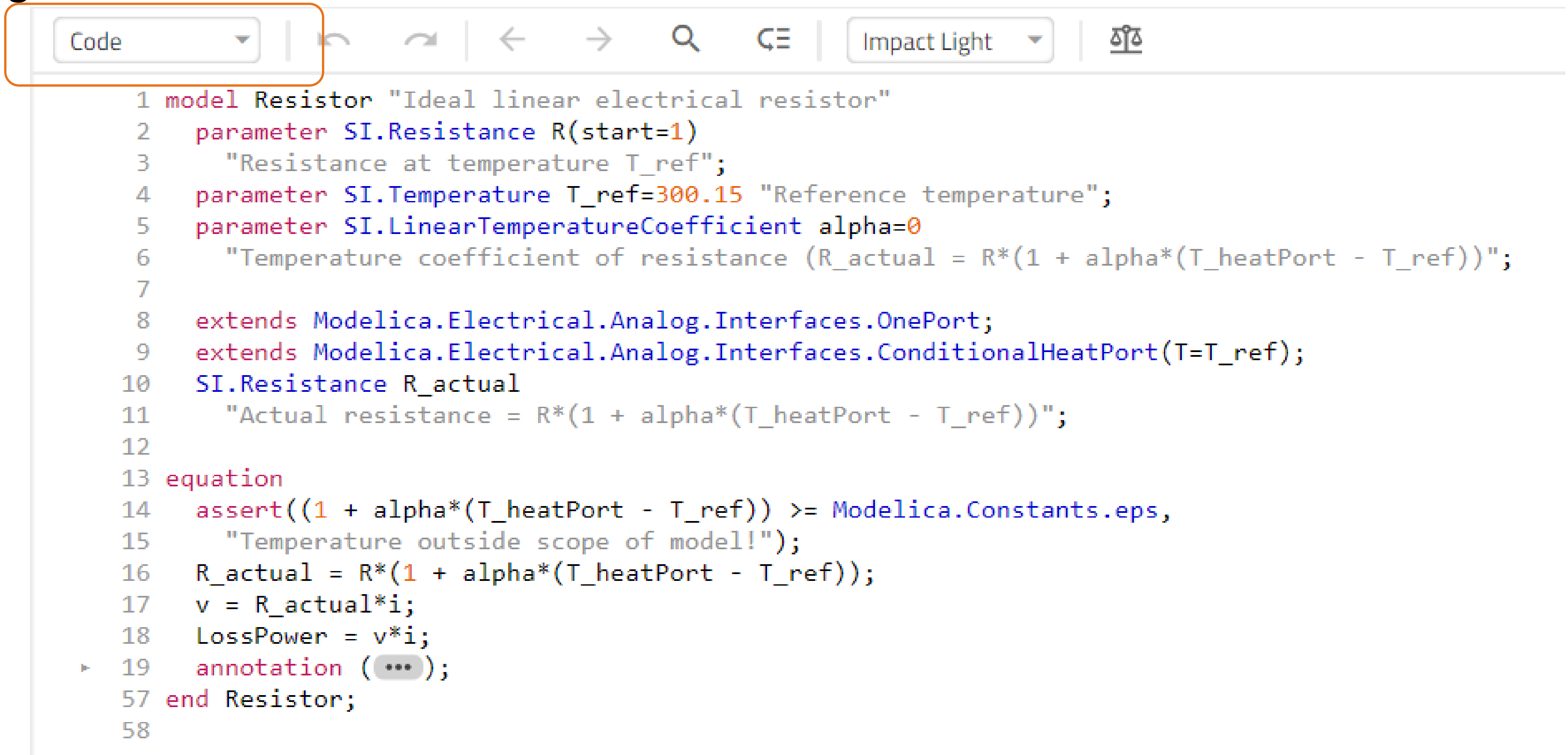
- *1998 by Christoph Clauss*
  initially implemented

For a model, documentation should contain the experimental frame!

ACCESSING THE SOURCE CODE

SOURCE CODE EDITOR

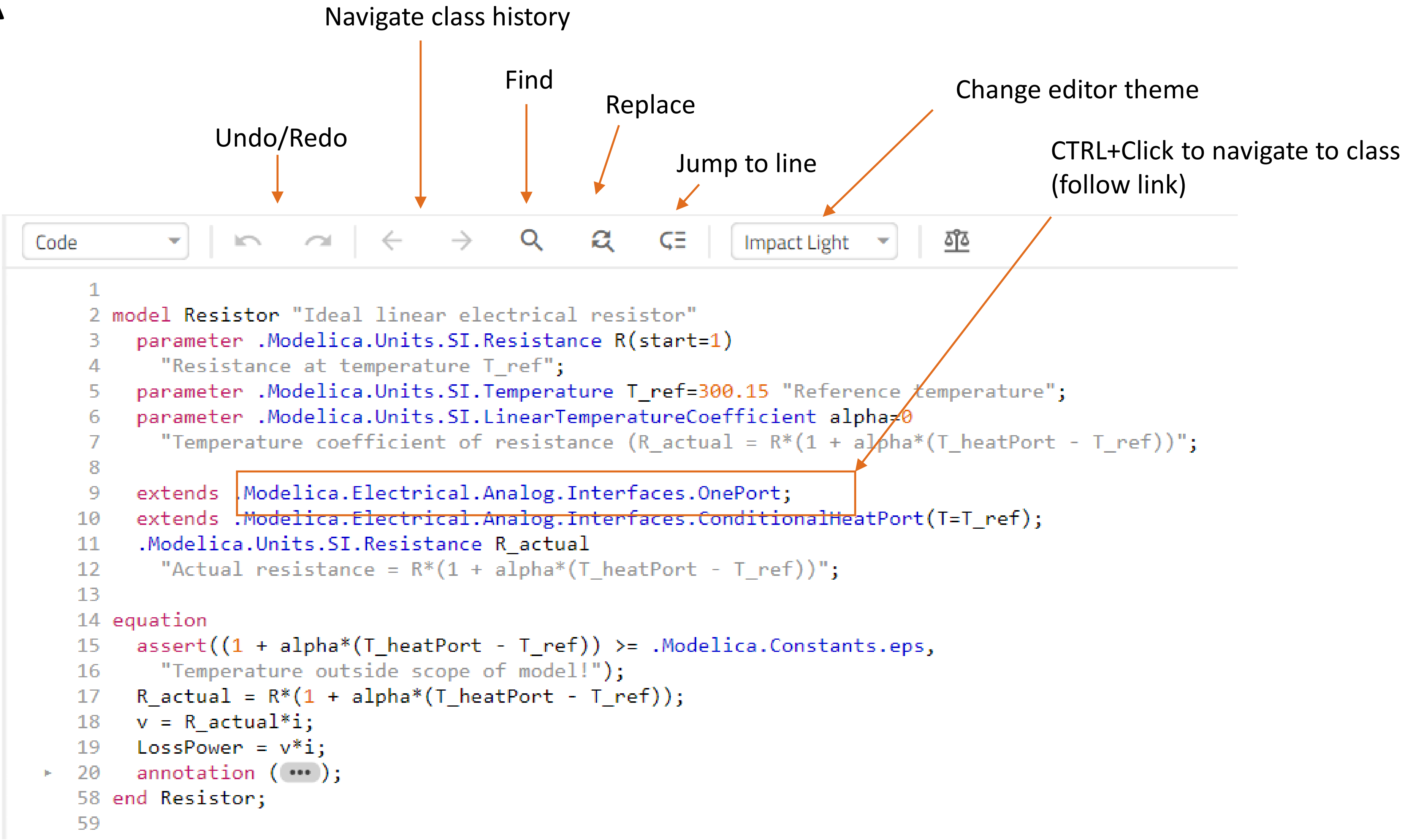
- Modelon Impact provides a code editor to edit the Modelica source code
- Accessed through toolbar:



```
model Resistor "Ideal linear electrical resistor"
  parameter SI.Resistance R(start=1)
    "Resistance at temperature T_ref";
  parameter SI.Temperature T_ref=300.15 "Reference temperature";
  parameter SI.LinearTemperatureCoefficient alpha=0
    "Temperature coefficient of resistance (R_actual = R*(1 + alpha*(T_heatPort - T_ref))";

  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  extends Modelica.Electrical.Analog.Interfaces.ConditionalHeatPort(T=T_ref);
  SI.Resistance R_actual
    "Actual resistance = R*(1 + alpha*(T_heatPort - T_ref))";

equation
  assert((1 + alpha*(T_heatPort - T_ref)) >= Modelica.Constants.eps,
    "Temperature outside scope of model!");
  R_actual = R*(1 + alpha*(T_heatPort - T_ref));
  v = R_actual*i;
  LossPower = v*i;
  annotation (•••);
end Resistor;

```

CODE EDITOR

- Basic functionality
- Includes syntax check
- Syntax needs to be ok to be able to save changes.



Navigate class history

Find

Replace

Undo/Redo

Jump to line

Change editor theme

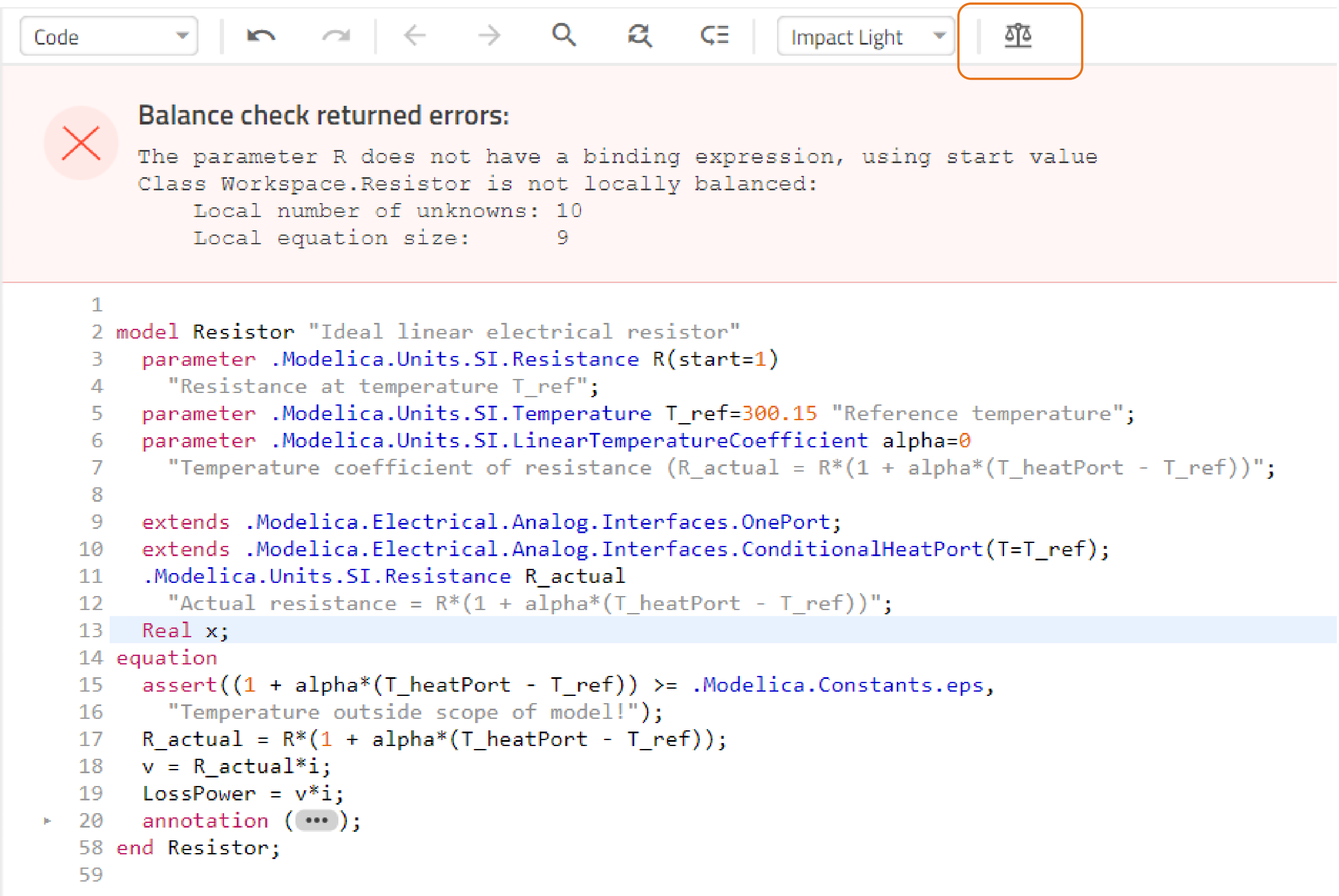
CTRL+Click to navigate to class (follow link)

```

model Resistor "Ideal linear electrical resistor"
  parameter .Modelica.Units.SI.Resistance R(start=1)
    "Resistance at temperature T_ref";
  parameter .Modelica.Units.SI.Temperature T_ref=300.15 "Reference temperature";
  parameter .Modelica.Units.SI.LinearTemperatureCoefficient alpha=0
    "Temperature coefficient of resistance (R_actual = R*(1 + alpha*(T_heatPort - T_ref))";

  extends .Modelica.Electrical.Analog.Interfaces.OnePort;
  extends .Modelica.Electrical.Analog.Interfaces.ConditionalHeatPort(T=T_ref);
  .Modelica.Units.SI.Resistance R_actual
    "Actual resistance = R*(1 + alpha*(T_heatPort - T_ref))";

equation
  assert((1 + alpha*(T_heatPort - T_ref)) >= .Modelica.Constants.eps,
    "Temperature outside scope of model!");
  R_actual = R*(1 + alpha*(T_heatPort - T_ref));
  v = R_actual*i;
  LossPower = v*i;
  annotation (•••);
end Resistor;

```

Copyright © 2022 Modelon

LOCAL BALANCE CHECK

- Checks semantics, and local balance of variables and equations



Balance check returned errors:

```
The parameter R does not have a binding expression, using start value
Class Workspace.Resistor is not locally balanced:
    Local number of unknowns: 10
    Local equation size:       9
```

```

model Resistor "Ideal linear electrical resistor"
  parameter .Modelica.Units.SI.Resistance R(start=1)
    "Resistance at temperature T_ref";
  parameter .Modelica.Units.SI.Temperature T_ref=300.15 "Reference temperature";
  parameter .Modelica.Units.SI.LinearTemperatureCoefficient alpha=0
    "Temperature coefficient of resistance (R_actual = R*(1 + alpha*(T_heatPort - T_ref))";

  extends .Modelica.Electrical.Analog.Interfaces.OnePort;
  extends .Modelica.Electrical.Analog.Interfaces.ConditionalHeatPort(T=T_ref);
  .Modelica.Units.SI.Resistance R_actual
    "Actual resistance = R*(1 + alpha*(T_heatPort - T_ref))";
  Real x;
equation
  assert((1 + alpha*(T_heatPort - T_ref)) >= .Modelica.Constants.eps,
    "Temperature outside scope of model!");
  R_actual = R*(1 + alpha*(T_heatPort - T_ref));
  v = R_actual*i;
  LossPower = v*i;
  annotation (•••);
end Resistor;

```

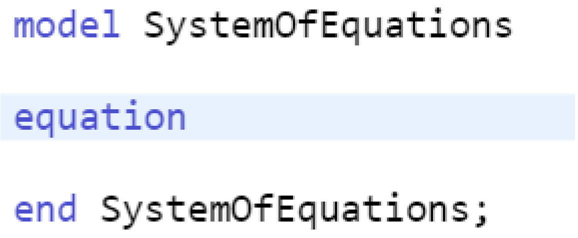
MODELICA AT A GLANCE

MODELICA AT A GLANCE

- Object-oriented → signalVoltage.V
- Acausal → U = R*I (no need to provide variants I := U/R or R = U/I)
- Type-based → Real vs Integer vs Boolean vs Modelica.SIunits.Voltage
- More complex structures such as partial, replaceable model etc.

The model class is divided into two main sections

1. before the ***equation*** keyword
   contains all component, parameters, inputs, outputs and variable declarations
2. after the ***equation*** keyword
   contains all equations and connects



```
model SystemOfEquations

equation

end SystemOfEquations;
```

VARIABLES AND TYPES

VARIABLE DECLARATIONS

- Available default variable types:

Real	floating point variable, e.g. 1.0, -2.3e-5
Integer	integer variable, e.g. 1, 4, -333
Boolean	boolean variable, e.g. false, true
String	string, e.g. "from file:“

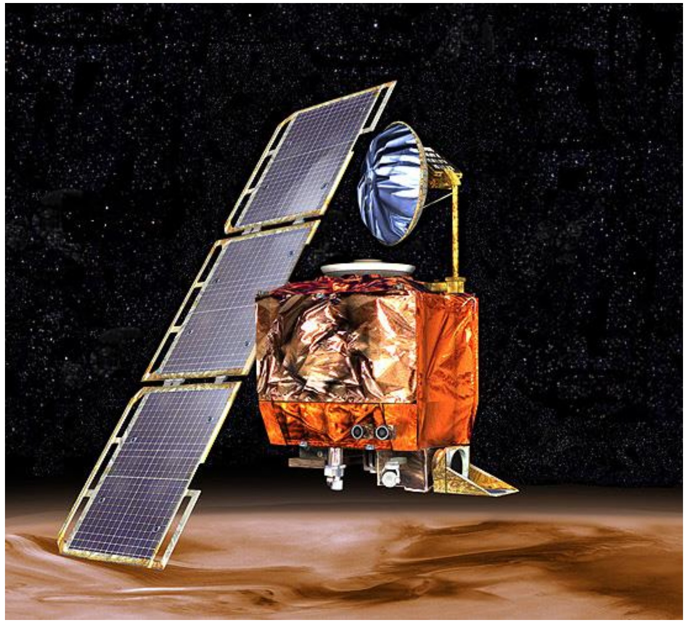
- Attributes of Real variables:

quantity	type of physical quantity
unit	unit used in equations
displayUnit	used in dialogs and postprocessing
min	minimal value of quantity
max	maximum value of quantity
nominal	used for scaling in numerical routines

WHY FOCUS ON UNITS?

- Mars Climate Orbiter Failure Board Release Report, Nov. 10, 1999:

  "The 'root cause' of the loss of the spacecraft was the failed ***translation*** *of* ***English units*** *into* ***metric units*** *in a segment of ground-based, navigation-related mission software, as NASA has previously announced," said Arthur Stephenson, chairman of the Mars Climate Orbiter Mission Failure Investigation Board.*

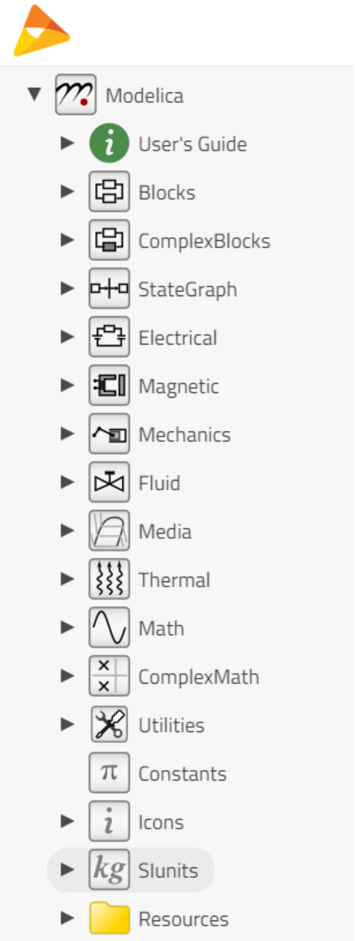


Ref:

https://www.jpl.nasa.gov/missions/mars-climate-orbiter

VARIABLE DECLARATIONS

- Modelica.SIunits contains all 450 ISO-standard units as predefined variable types.

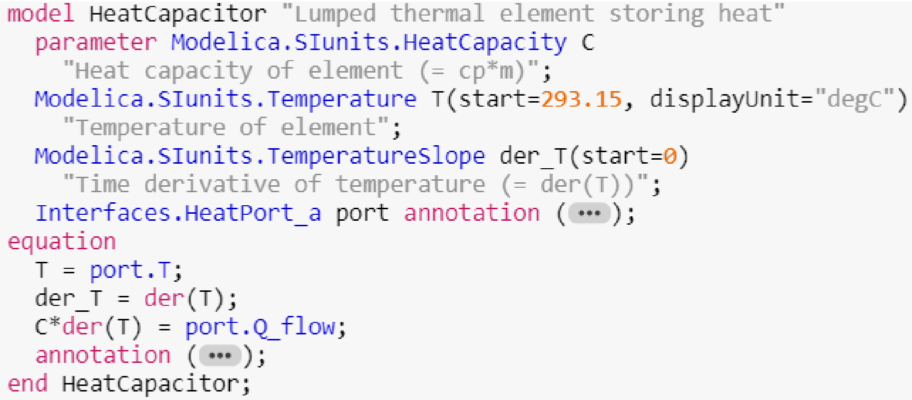


```
  <strong>parameter</strong> SI.Angle           phi = from_deg(180);   // convert 180 degree to radian
  <strong>parameter</strong> SI.AngularVelocity w   = from_rpm(3600);  // convert 3600 revolutions per
                                                       // to radian per seconds
</pre>

</html>"));
  end Conversions;

  // Space and Time (chapter 1 of ISO 31-1992)

  type Angle = Real (
      final quantity="Angle",
      final unit="rad",
      displayUnit="deg");
  type SolidAngle = Real (final quantity="SolidAngle", final unit="sr");
  type Length = Real (final quantity="Length", final unit="m");
  type PathLength = Length;
  type Position = Length;
  type Distance = Length (min=0);
  type Breadth = Length(min=0);
  type Height = Length(min=0);
  type Thickness = Length(min=0);
  type Radius = Length(min=0);
  type Diameter = Length(min=0);
  type Area = Real (final quantity="Area", final unit="m2");
  type Volume = Real (final quantity="Volume", final unit="m3");
  type Time = Real (final quantity="Time", final unit="s");
  type Duration = Time;
  type AngularVelocity = Real (
      final quantity="AngularVelocity",
      final unit="rad/s");
  type AngularAcceleration = Real (final quantity="AngularAcceleration", final unit=
           "rad/s2");
  type Velocity = Real (final quantity="Velocity", final unit="m/s");
  type Acceleration = Real (final quantity="Acceleration", final unit="m/s2");

  // Periodic and related phenomens (chapter 2 of ISO 31-1992)
  type Period = Real (final quantity="Time", final unit="s");
  type Frequency = Real (final quantity="Frequency", final unit="Hz");
  type AngularFrequency = Real (final quantity="AngularFrequency", final unit=
        "rad/s");
  type Wavelength = Real (final quantity="Wavelength", final unit="m");
  type Wavelenght = Wavelength;
  // For compatibility reasons only
  type WaveNumber = Real (final quantity="WaveNumber", final unit="m-1");
  type CircularWaveNumber = Real (final quantity="CircularWaveNumber", final unit=
           "rad/m");
  type AmplitudeLevelDifference = Real (final quantity=
        "AmplitudeLevelDifference", final unit="dB");
  type PowerLevelDifference = Real (final quantity="PowerLevelDifference",
       final unit="dB");
  type DampingCoefficient = Real (final quantity="DampingCoefficient", final unit=
           "s-1");
```

Copyright © 2022 Modelon

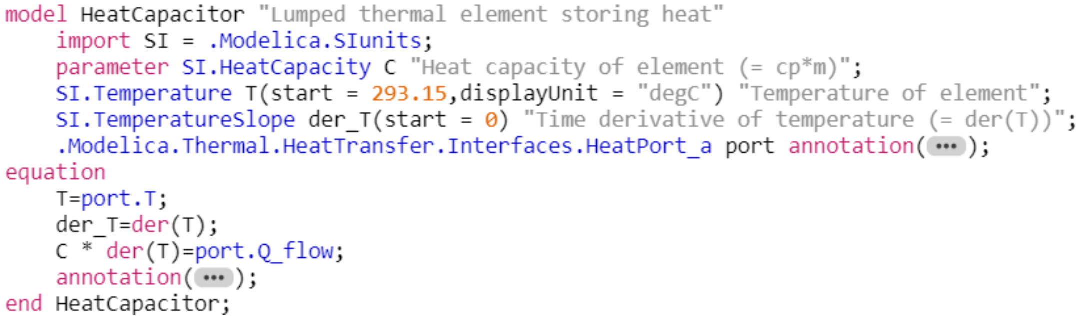
VARIABLE DECLARATIONS

- Declarations using SI units package:



```
model HeatCapacitor "Lumped thermal element storing heat"
  parameter Modelica.SIunits.HeatCapacity C
    "Heat capacity of element (= cp*m)";
  Modelica.SIunits.Temperature T(start=293.15, displayUnit="degC")
    "Temperature of element";
  Modelica.SIunits.TemperatureSlope der_T(start=0)
    "Time derivative of temperature (= der(T))";
  Interfaces.HeatPort_a port annotation (•••);
equation
  T = port.T;
  der_T = der(T);
  C*der(T) = port.Q_flow;
  annotation (•••);
end HeatCapacitor;
```

- In order to avoid repeating the package name at each declaration, create a package shortcut:

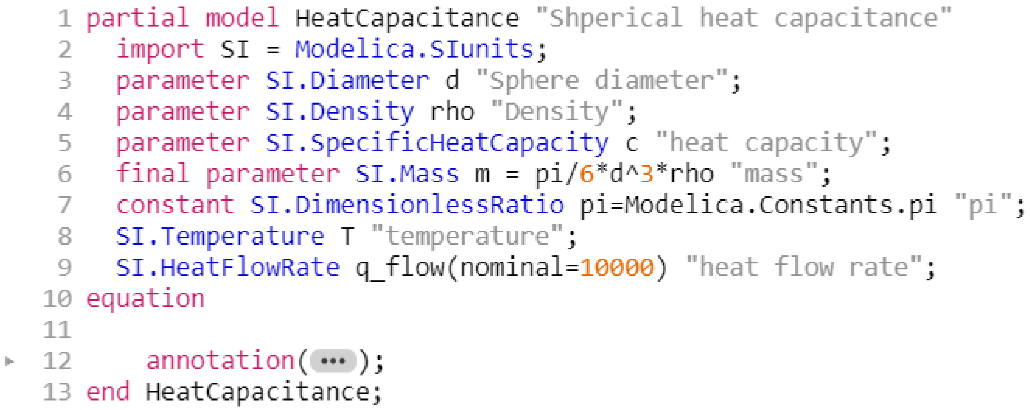


```
model HeatCapacitor "Lumped thermal element storing heat"
    import SI = .Modelica.SIunits;
    parameter SI.HeatCapacity C "Heat capacity of element (= cp*m)";
    SI.Temperature T(start = 293.15,displayUnit = "degC") "Temperature of element";
    SI.TemperatureSlope der_T(start = 0) "Time derivative of temperature (= der(T))";
    .Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port annotation(•••);
equation
    T=port.T;
    der_T=der(T);
    C * der(T)=port.Q_flow;
    annotation(•••);
end HeatCapacitor;
```

VARIABLE DECLARATIONS

- Time variability is set with a variable prefix:

no prefix	variable	can change with time
`parameter`	parameter	constant with time, may be modified
`constant`	constant	constant with time, may not be modified

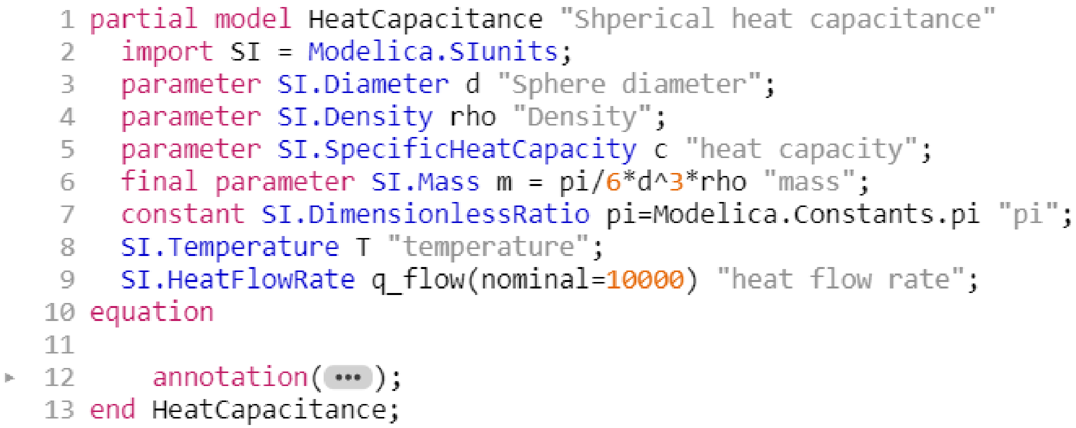


```
partial model HeatCapacitance "Shperical heat capacitance"
  import SI = Modelica.SIunits;
  parameter SI.Diameter d "Sphere diameter";
  parameter SI.Density rho "Density";
  parameter SI.SpecificHeatCapacity c "heat capacity";
  final parameter SI.Mass m = pi/6*d^3*rho "mass";
  constant SI.DimensionlessRatio pi=Modelica.Constants.pi "pi";
  SI.Temperature T "temperature";
  SI.HeatFlowRate q_flow(nominal=10000) "heat flow rate";
equation

    annotation(•••);
end HeatCapacitance;
```

- In a component of the model above only **d**, **rho** and **c** can be modified at the container level
- A default equation may be added for the parameter declarations

VARIABLE DECLARATION

- The model shall describe a spherical capacitance. Its mass is computed from diameter **`d`** and density **`rho`**:



```
partial model HeatCapacitance "Shperical heat capacitance"
  import SI = Modelica.SIunits;
  parameter SI.Diameter d "Sphere diameter";
  parameter SI.Density rho "Density";
  parameter SI.SpecificHeatCapacity c "heat capacity";
  final parameter SI.Mass m = pi/6*d^3*rho "mass";
  constant SI.DimensionlessRatio pi=Modelica.Constants.pi "pi";
  SI.Temperature T "temperature";
  SI.HeatFlowRate q_flow(nominal=10000) "heat flow rate";
equation

    annotation(•••);
end HeatCapacitance;
```

- In order to prevent it from appearing in the parameter dialog of the corresponding component and being modified with an inconsistent value, m received the final prefix
- The variables used in the default equation must not have a higher variability than the declared variable itself, i.e., for parameters only parameters and constants are allowed

ARRAYS AND MATRICES

ARRAYS AND MATRICES

Declaration of multidimensional arrays:

```
parameter Real v1[3] = {1,2,3};
```

{} is array constructor of arrays with arbitrary dimension.

```
parameter Real v2[3,1] = [1;2;3];
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

[ ] generates matrices, and acts as concatenation operator.

```
parameter Real m1[2,3] = {{11,12,13}, {21,22,23}};
```

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

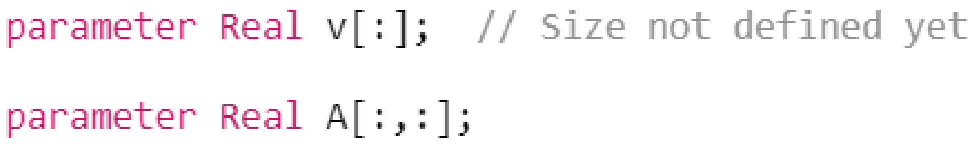
```
parameter Real m2[2,3] = [11, 12, 13; 21, 22, 23];
```

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

```
parameter Real m3[3,3] = [m1; transpose([v])];
```

ARRAYS AND MATRICES

“:” in the declaration section is used, when size of the array is undefined

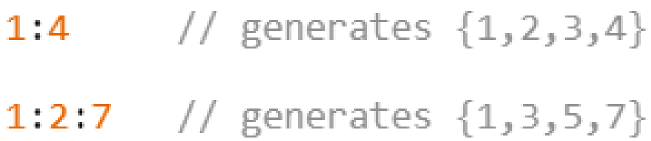


```
parameter Real v[:];  // Size not defined yet

parameter Real A[:,:];
```

Access to matrix elements:

```
M2[2,3]  // element [2,3] of Matrix M2
```

Vector constructor normally used to generate an indices-vector:



```
1:4      // generates {1,2,3,4}

1:2:7    // generates {1,3,5,7}
```

Extraction mechanism of sub-matrices as in Matlab:

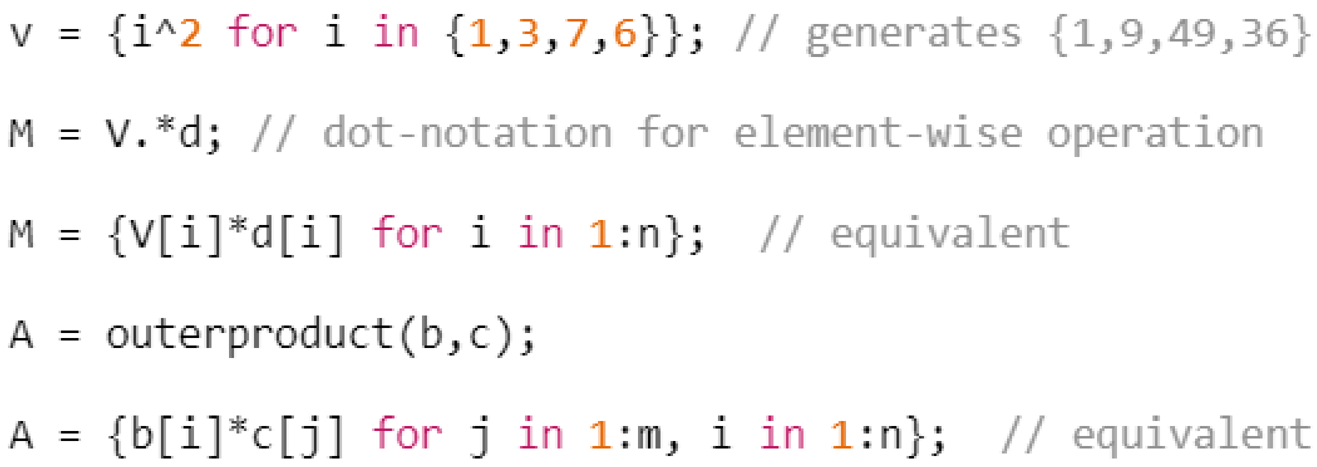
```
M2[2:4,3]   // generates {M2[2,3], M2[3,3], M2[4,3]}
```

ARRAYS AND MATRICES

- Operations according to standard mathematics. Compared to e.g. Matlab, it is worth noticing that Modelica handles physical vectors, i.e.:

  vector*vector = scalar [for ex:{1,2,3}*{4,5,6} = 32]
  matrix*vector = vector [for ex:[1,2,3;4,5,6]*{1,2,3}={14, 32}]
  vector*matrix = vector [for ex:{1,2,3}*[4;5;6]={32}]
  matrix*matrix = matrix [for ex:[1;2]*[3,4]=[3, 4;6, 8]]

- Creating an array of a general expression with an array constructor:

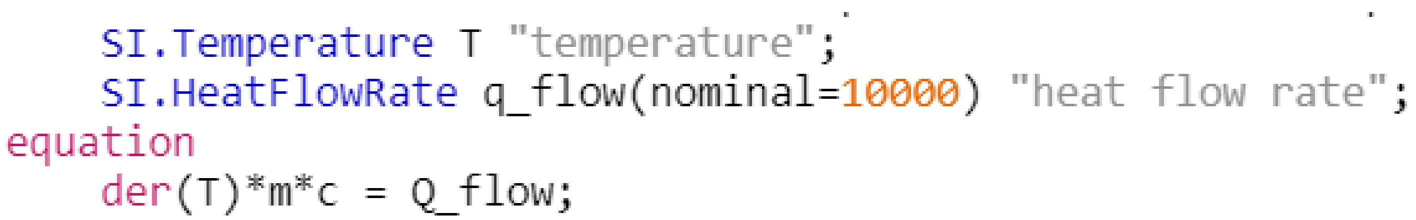


```
v = {i^2 for i in {1,3,7,6}}; // generates {1,9,49,36}

M = V.*d; // dot-notation for element-wise operation

M = {V[i]*d[i] for i in 1:n};  // equivalent

A = outerproduct(b,c);

A = {b[i]*c[j] for j in 1:m, i in 1:n};  // equivalent
```

EQUATIONS AND ALGORITHMS

EQUATIONS

- Equations are added in the equation section after the `equation` keyword
- `time` is a global built-in variable
- Differential equations are expressed with the `der`-operator. It denotes the time derivative of the expression:

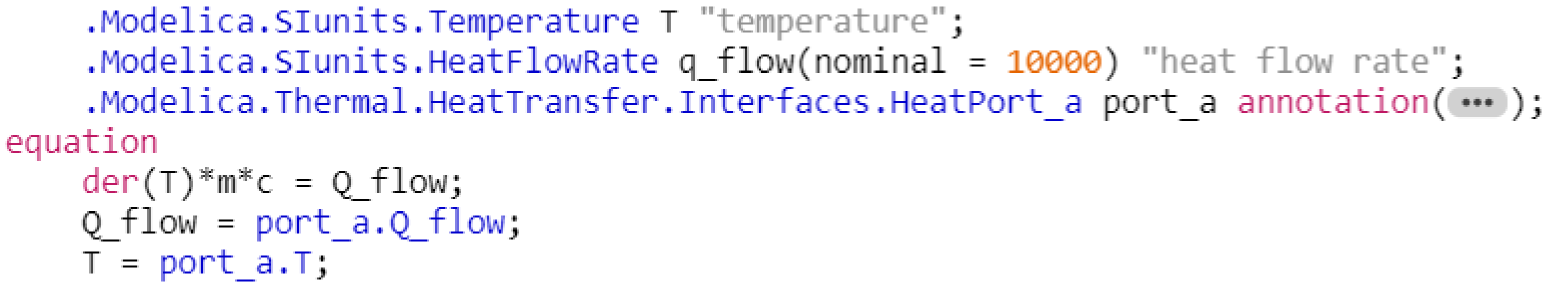


```
    SI.Temperature T "temperature";
    SI.HeatFlowRate q_flow(nominal=10000) "heat flow rate";
equation
    der(T)*m*c = Q_flow;
```

- ***Order of equations*** and which of the variables are located on the ***left or right-hand side*** of the equality sign is ***irrelevant***

INCLUDING CONNECTORS

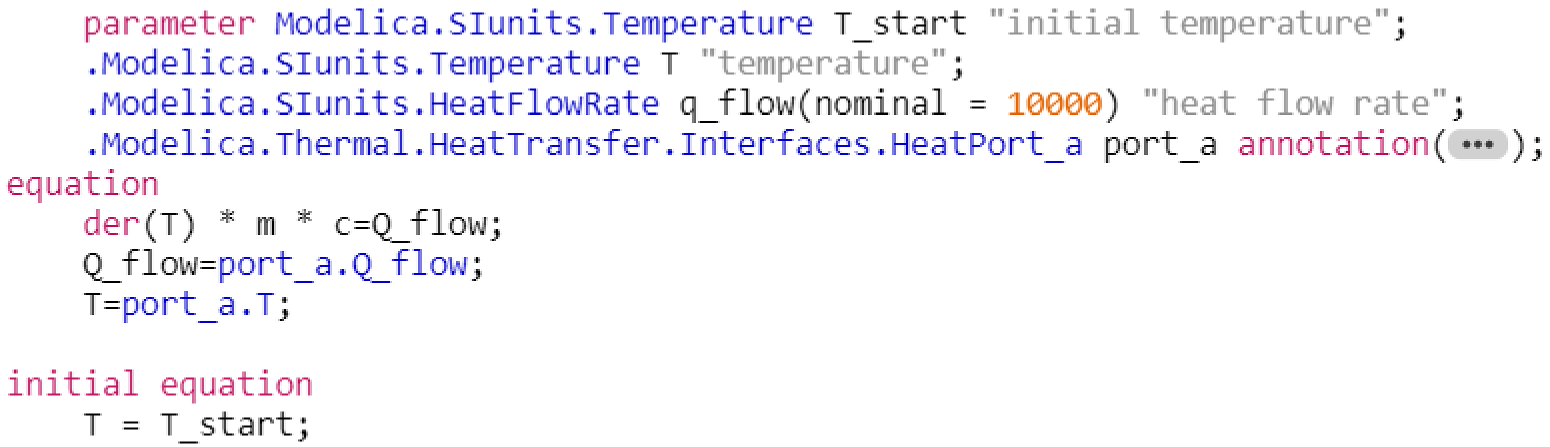
- The example model has two time-varying variables and just one equation. The heat flow rate is determined outside the capacitance and shall cross its boundary through a connector.
  1. On the modeling canvas, drag in a component of the connector class **`Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a`**
  2. Create a relationship between connector variables and variables declared in the model



```
    .Modelica.SIunits.Temperature T "temperature";
    .Modelica.SIunits.HeatFlowRate q_flow(nominal = 10000) "heat flow rate";
    .Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a annotation(•••);
equation
    der(T)*m*c = Q_flow;
    Q_flow = port_a.Q_flow;
    T = port_a.T;
```

INITIALIZATION

- Integrated variables require a start value, to solve the initial time problem
- They are set in an **`initial equation`** section, which is only evaluated at **`time`** = 0



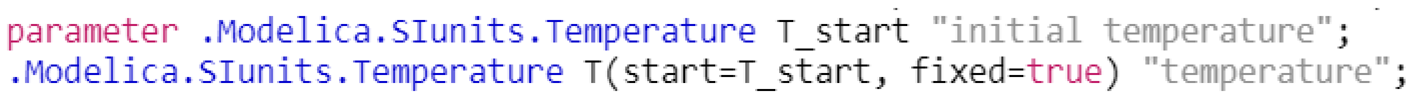
```
    parameter Modelica.SIunits.Temperature T_start "initial temperature";
    .Modelica.SIunits.Temperature T "temperature";
    .Modelica.SIunits.HeatFlowRate q_flow(nominal = 10000) "heat flow rate";
    .Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a annotation(...);
equation
    der(T) * m * c=Q_flow;
    Q_flow=port_a.Q_flow;
    T=port_a.T;

initial equation
    T = T_start;
```

- It is common practice to introduce a start value parameter, here **`T_start`**, which can be propagated through the component levels.

INITIALIZATION, FIXED =TRUE/FALSE

- Alternative way to set start values:

  set start-attribute in variable declaration



```
parameter .Modelica.SIunits.Temperature T_start "initial temperature";
.Modelica.SIunits.Temperature T(start=T_start, fixed=true) "temperature";
```

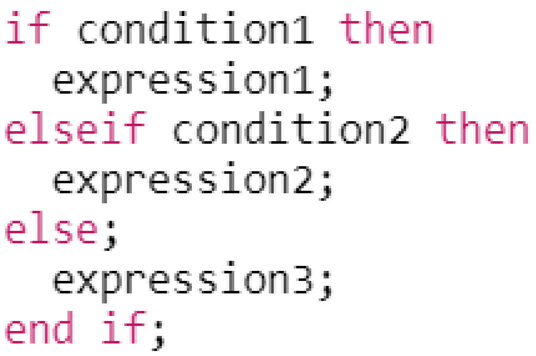
- Difference to initial equations:
  - the start-attribute value is only used as an initial value of variable **T** if this is a **state variable,** i.e. an integration variable in the numerical solver. By setting `fixed=true` it generates an initial equation.
  - if the start-attribute is set for an **algebraic variable**, this value may be used by the solver as a **guess value** for iteration variables in non-linear initial equations.
  - The number of equations in the initial equation section is restricted to the degrees of freedom of the simulation experiment, i.e. the number of **state variables**. (Same holds for the number of `fixed=true`)

ALGORITHMS

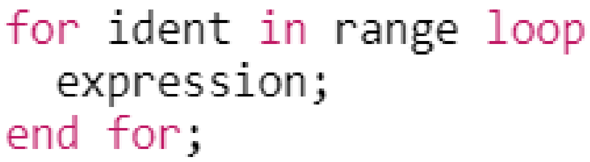
- The standard way in Modelica is to write equations, but also algorithms can be used, instead of ‘equation’ use ‘algorithm’ and instead of ‘=’ use ‘:=‘.
- Algorithms are treated as a sequence of assignments.
- It is possible to use both an algorithm section and equation sections in the same model, but maximum one algorithm section.
- A value that is not assigned in an algorithm is assumed to be zero. There is no error message!

IF STATEMENTS / FOR LOOPS

- The syntax is (same for both algorithm and equation)



```
if condition1 then
  expression1;
elseif condition2 then
  expression2;
else;
  expression3;
end if;
```



```
for ident in range loop
  expression;
end for;
```

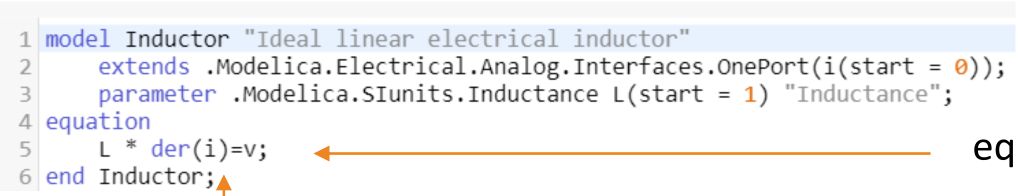
NOTE: in equation sections, number of variables and equations must match, so no overwriting is possible.

OPERATORS AND STATEMENTS

DEVELOPING MODELS BY CODING

Equations are written in the `equation` section

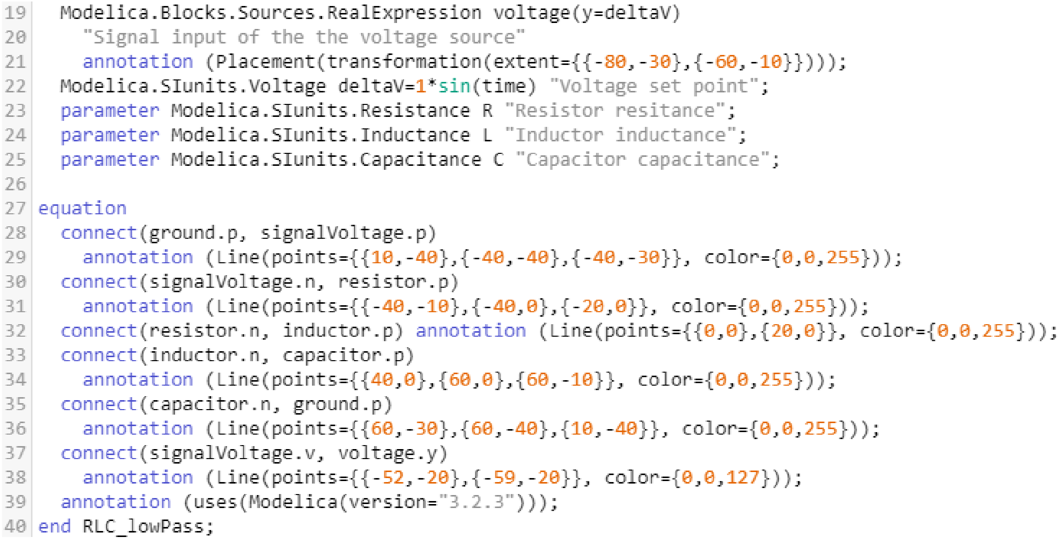
- `der()` → operator to indicate a time derivative of a variable
- `connect()` → operator to describe a connection between connectors
  `connect()` corresponds to a set of equations (discussed in part 2)
- `if`-, `when`-, `while`-statements etc. can be included



```
model Inductor "Ideal linear electrical inductor"
    extends .Modelica.Electrical.Analog.Interfaces.OnePort(i(start = 0));
    parameter .Modelica.SIunits.Inductance L(start = 1) "Inductance";
equation
    L * der(i)=v;
end Inductor;
```

derivative of current i

equation



```
  Modelica.Blocks.Sources.RealExpression voltage(y=deltaV)
    "Signal input of the the voltage source"
    annotation (Placement(transformation(extent={{-80,-30},{-60,-10}})));
  Modelica.SIunits.Voltage deltaV=1*sin(time) "Voltage set point";
  parameter Modelica.SIunits.Resistance R "Resistor resitance";
  parameter Modelica.SIunits.Inductance L "Inductor inductance";
  parameter Modelica.SIunits.Capacitance C "Capacitor capacitance";

equation
  connect(ground.p, signalVoltage.p)
    annotation (Line(points={{10,-40},{-40,-40},{-40,-30}}, color={0,0,255}));
  connect(signalVoltage.n, resistor.p)
    annotation (Line(points={{-40,-10},{-40,0},{-20,0}}, color={0,0,255}));
  connect(resistor.n, inductor.p) annotation (Line(points={{0,0},{20,0}}, color={0,0,255}));
  connect(inductor.n, capacitor.p)
    annotation (Line(points={{40,0},{60,0},{60,-10}}, color={0,0,255}));
  connect(capacitor.n, ground.p)
    annotation (Line(points={{60,-30},{60,-40},{10,-40}}, color={0,0,255}));
  connect(signalVoltage.v, voltage.y)
    annotation (Line(points={{-52,-20},{-59,-20}}, color={0,0,127}));
  annotation (uses(Modelica(version="3.2.3")));
end RLC_lowPass;
```

CONNECTORS AND CONNECT()

CONNECTORS

- To make the components interact with each other, we need clear interfaces: Connectors
- Key for the connector concept is the difference between potential and flow variables
- Each component must have a set of equations that uniquely define its behavior based on its interfaces and initial conditions.
- The components and the connectors need to be balanced, i.e. the number of unknowns and equations must match.
- In a Modelica connector, a variable with the **flow** prefix is a flow variable, and a variable without a prefix is a potential variable.

POTENTIAL AND FLOW VARIABLES

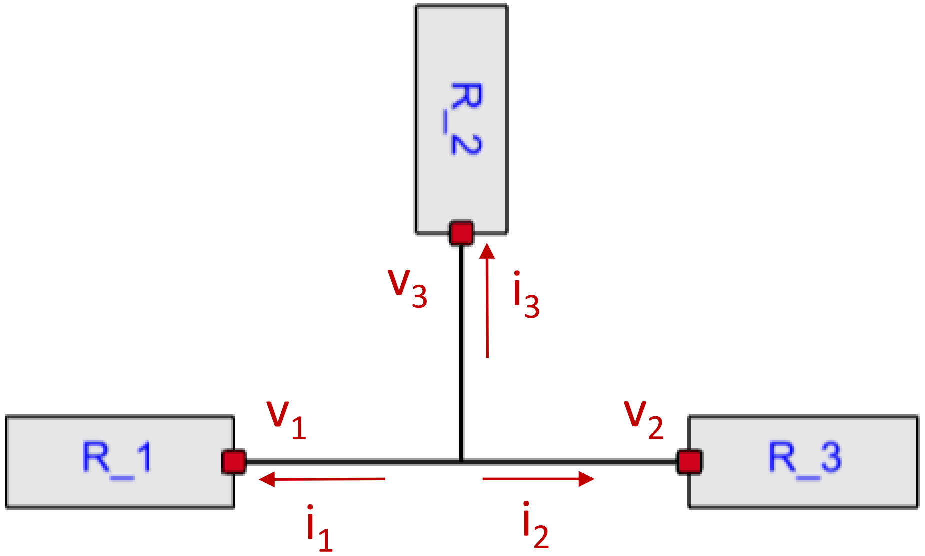
Electrical

Connector:

```
connector Pin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation(•••);
end Pin;
```

Modelica:

```
connect(R_1.pin,R_2.pin) annotation(•••);
connect(R_1.pin,R_3.pin) annotation(•••);
```



$$\boxed{v_1 = v_2 = v_3}$$

Potential: n-1 equations per connection set

$$\boxed{i_1 + i_2 + i_3 = 0}$$

Flow: 1 equation per connection set

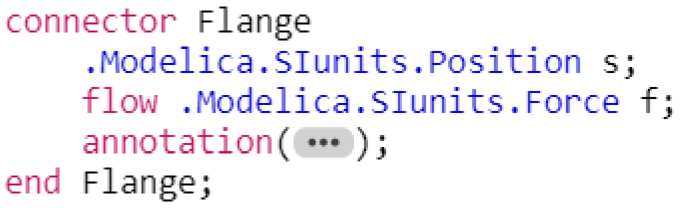
Generated equations from connection:

```
R_1.pin.v = R_2.pin.v
R_1.pin.v = R_3.pin.v
R_1.pin.i + R_2.pin.i + R_3.pin.i = 0
```

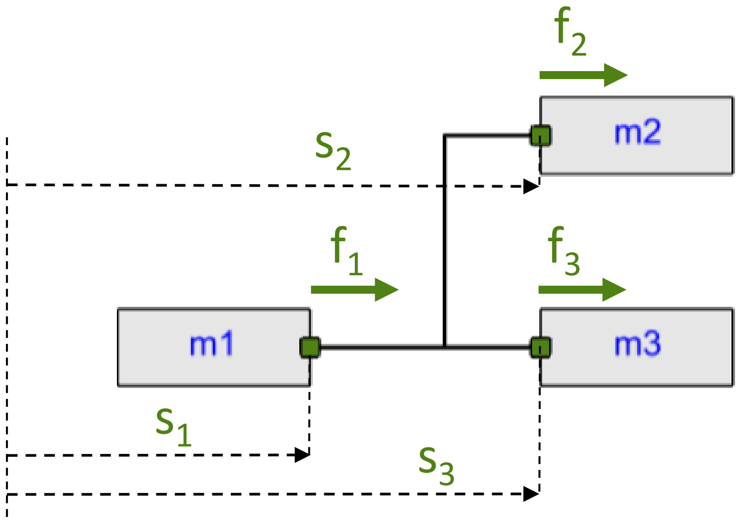
POTENTIAL AND FLOW VARIABLES

1D-mechanical

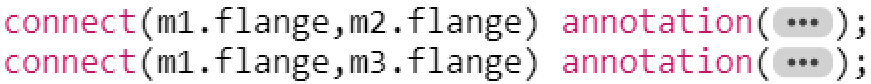
Connector:



```
connector Flange
    .Modelica.SIunits.Position s;
    flow .Modelica.SIunits.Force f;
    annotation(•••);
end Flange;
```



Modelica:



```
connect(m1.flange,m2.flange) annotation(•••);
connect(m1.flange,m3.flange) annotation(•••);
```

$$\boxed{s_1 = s_2 = s_3}$$

Potential: n-1 equations per connection set

$$\boxed{f_1 + f_2 + f_3 = 0}$$

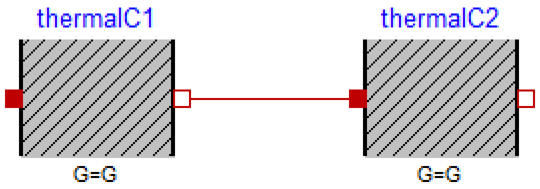
Flow: 1 equation per connection set

Generated equations from connection:

```
m1.flange.s = m2.flange.s
m1.flange.s = m3.flange.s
m1.flange.f + m2.flange.f + m3.flange.f = 0
```

POTENTIAL AND FLOW VARIABLES

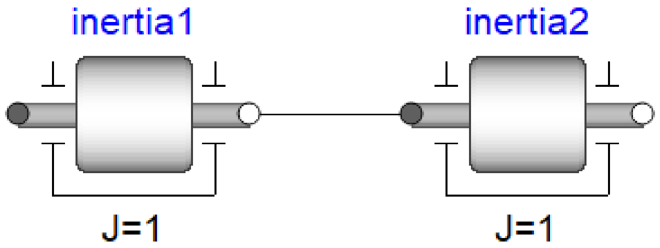
Heat transfer



```
connector HeatPort
Temperature T;
flow HeatFlowRate Q_flow;
end HeatPort;
```

```
thermalC1.heatPort_b.T = thermalC2.heatPort_a.T
thermalC1.heatPort_b.Q_flow + thermalC2.heatPort_a.Q_flow = 0
```

Rotational



```
connector Flange
Angle phi;
flow Torque tau;
end Flange;
```

```
inertia1.flange_a.phi = inertia2.flange_b.phi
inertia2.flange_a.tau + inertia2.flange_b.tau = 0
```

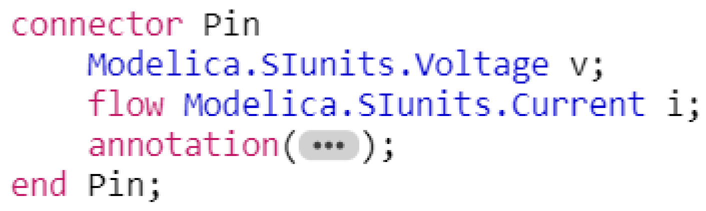
BALANCING CONCEPT AND PARTIAL

BALANCED MODELS

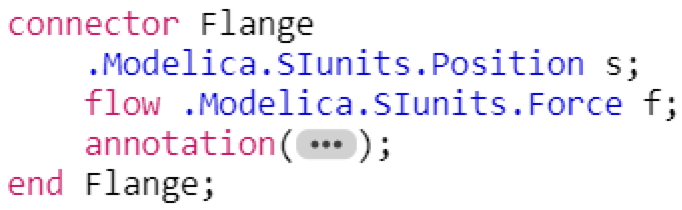
- A balanced model provides a set of equations so that either the flow or the potential variable can be solved for.
- In Modelica, all models should be “locally balanced”
- A model build from locally balanced sub-models is also balanced.
- This in turn requires that each connector has the same amount of flow and potential variables.

BALANCED MODELS

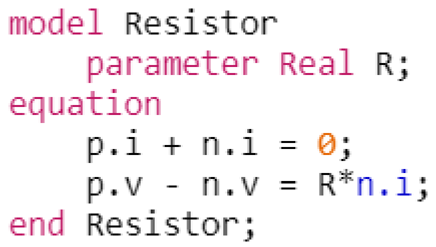
- In the examples below, each component has two connectors, each with one flow/potential variable. So two equations are needed, one for each flow/potential variable.



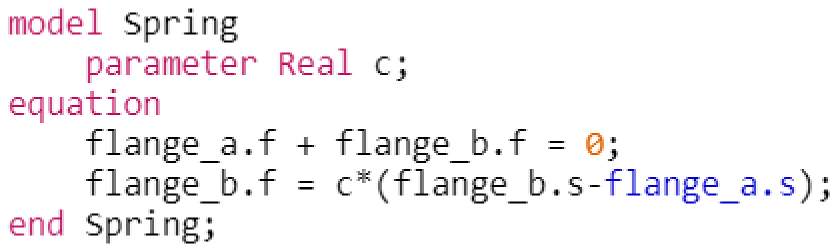
```
connector Pin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation(•••);
end Pin;
```



```
connector Flange
    .Modelica.SIunits.Position s;
    flow .Modelica.SIunits.Force f;
    annotation(•••);
end Flange;
```



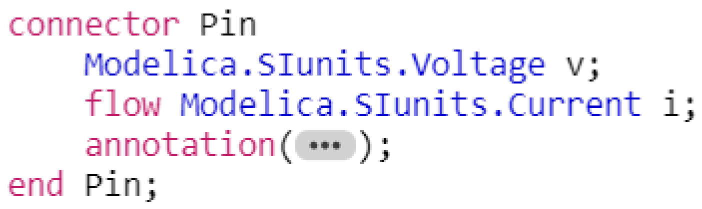
```
model Resistor
    parameter Real R;
equation
    p.i + n.i = 0;
    p.v - n.v = R*n.i;
end Resistor;
```



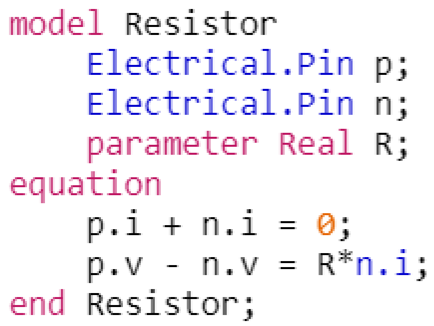
```
model Spring
    parameter Real c;
equation
    flange_a.f + flange_b.f = 0;
    flange_b.f = c*(flange_b.s-flange_a.s);
end Spring;
```

BALANCED MODELS

- How are the number of equations and unknown calculated?
- Model Resistor is balanced:



```
connector Pin
    Modelica.SIunits.Voltage v;
    flow Modelica.SIunits.Current i;
    annotation(•••);
end Pin;
```



```
model Resistor
    Electrical.Pin p;
    Electrical.Pin n;
    parameter Real R;
equation
    p.i + n.i = 0;
    p.v - n.v = R*n.i;
end Resistor;
```

```
Unknowns:
 p.i, p.v, n.i, n.v

Equations:
 p.i + n.i = 0;
 p.v - n.v = R*n.i;
```

+ 2 eqn for flow-variables
p.i and n.i,
when you check for local balance

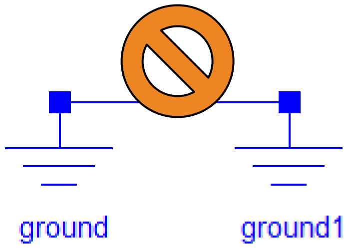
BALANCED MODELS

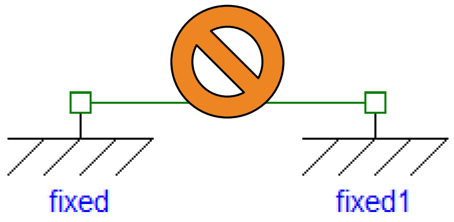
- Not all models have a relation between flow and potential, e.g. an electrical ground or a mechanical fixed:

```
model Ground
 ..
equation
 p.v = 0;
end Ground;
```

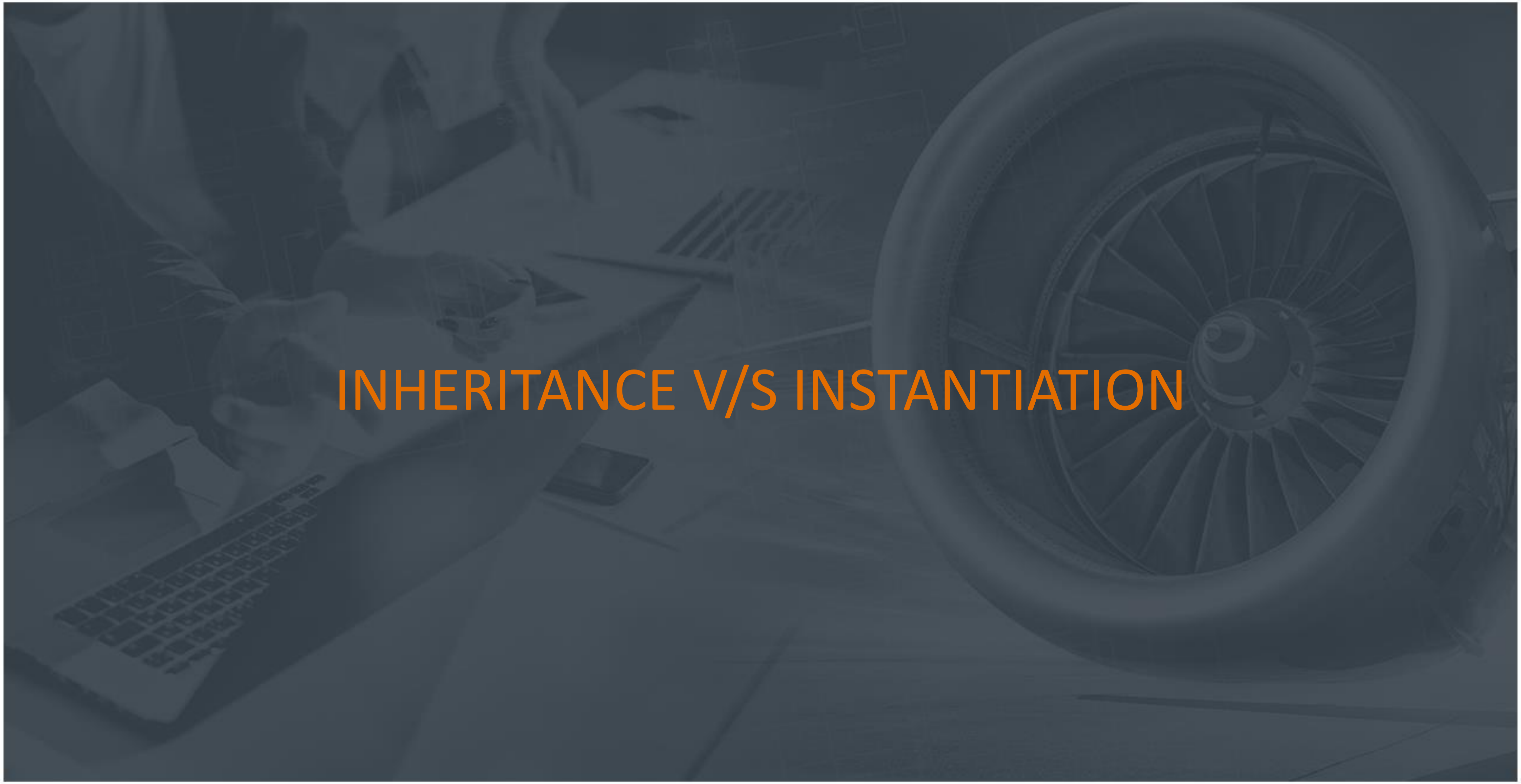
```
model Fixed
 ..
equation
 flange_a.s = 0;
end Fixed;
```

- As a result, these components cannot be connected to each other.





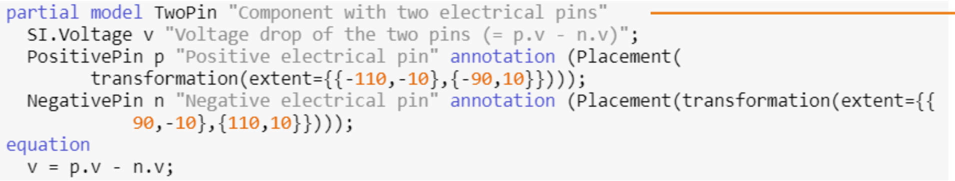
- The same holds for a model that directly supplies flow information, e.g. different types of sources.



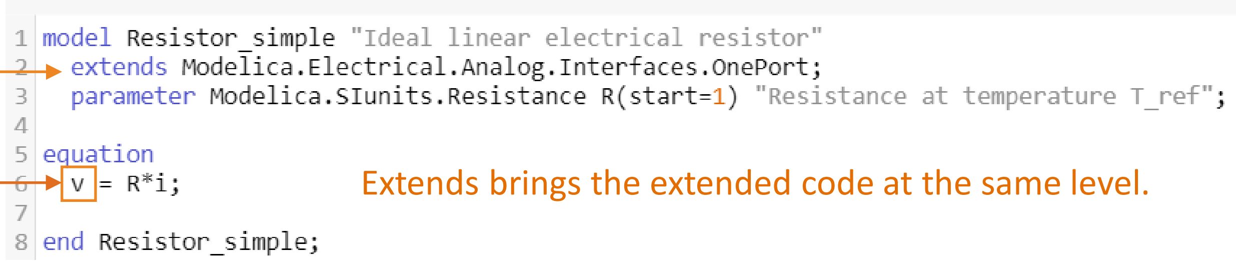
INHERITANCE V/S INSTANTIATION

OBJECT-ORIENTATION

- Extends



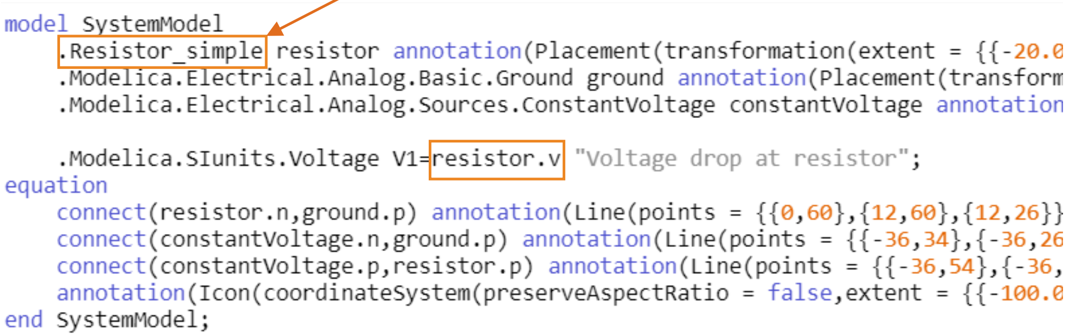
```
partial model TwoPin "Component with two electrical pins"
  SI.Voltage v "Voltage drop of the two pins (= p.v - n.v)";
  PositivePin p "Positive electrical pin" annotation (Placement(
        transformation(extent={{-110,-10},{-90,10}})));
  NegativePin n "Negative electrical pin" annotation (Placement(transformation(extent={{
          90,-10},{110,10}})));
equation
  v = p.v - n.v;
```



```
model Resistor_simple "Ideal linear electrical resistor"
  extends Modelica.Electrical.Analog.Interfaces.OnePort;
  parameter Modelica.SIunits.Resistance R(start=1) "Resistance at temperature T_ref";

equation
  v = R*i;

end Resistor_simple;
```

Extends brings the extended code at the same level.

- Instantiation



```
model SystemModel
    .Resistor_simple resistor annotation(Placement(transformation(extent = {{-20.0
    .Modelica.Electrical.Analog.Basic.Ground ground annotation(Placement(transform
    .Modelica.Electrical.Analog.Sources.ConstantVoltage constantVoltage annotation

    .Modelica.SIunits.Voltage V1=resistor.v "Voltage drop at resistor";
equation
    connect(resistor.n,ground.p) annotation(Line(points = {{0,60},{12,60},{12,26}}
    connect(constantVoltage.n,ground.p) annotation(Line(points = {{-36,34},{-36,26
    connect(constantVoltage.p,resistor.p) annotation(Line(points = {{-36,54},{-36,
    annotation(Icon(coordinateSystem(preserveAspectRatio = false,extent = {{-100.0
end SystemModel;
```

Instantiate brings the code one level below.
Accessed through dot notation.

Can be instantiated several times
(with different name).

Modelon

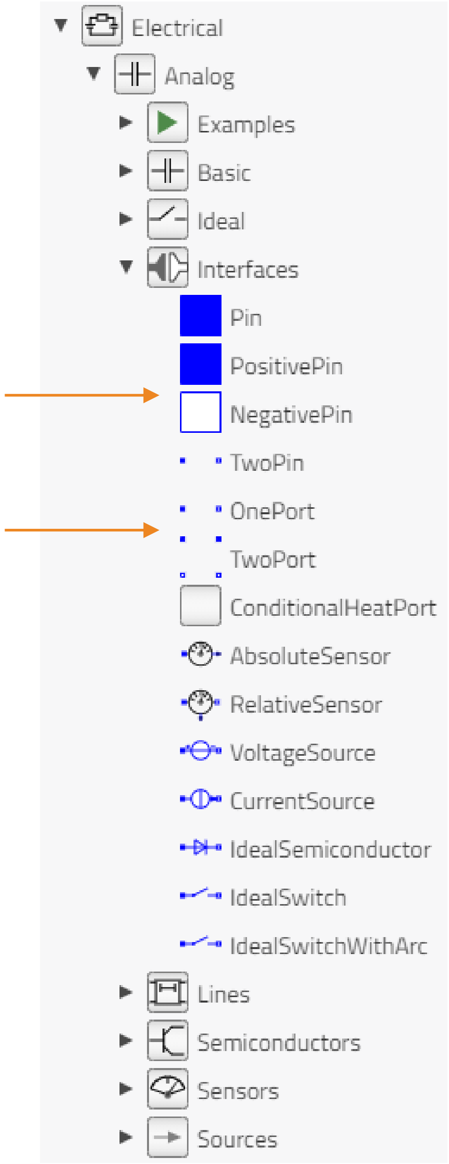
DEVELOPING MODELS BY CODING

Developing model using Modelica code

- Should be only when necessary (or for fun)
- Should reuse existing base classes as much as possible
- Requires much maintenance work and testing

Extension v/s Instantiation

- `extends Modelica.Electrical.Analog.Interfaces.TwoPin`
  - Model inherit all content from the extended classes (interfaces, equations, icon etc.)
  - All code from `TwoPin` will be inlined in the model
- `Modelica.Electrical.Analog.Sources.SignalVoltage signalVoltage`
  - Creates an instance of `SignalVoltage` called `signalVoltage`
  - Accessing variables of the instance through dot notation (e.g. `signalVoltage.v`)



`connector`

`partial model`
(base class or template)

OBJECT-ORIENTATION

When to do what?

- Extend:
  - Interfaces, templates, icons → common parts, partial
  - Physical effects → abstraction of a phenomena
  - Flat model and results
- Instantiate:
  - Models, Records, etc. → container / entity
  - “Physical components” → model composition representing user expectations
  - Structured model and results

WORKSHOP 3.1

In this workshop you will:

- Create a simple cake model