

Introducing the online reference

Welcome to the ICMS Online Reference. Topics covered include:

- [Architecture](#)
- [ICMS Terminology](#)
- [Working With Classes](#)
- [Working With Models](#)
- [Working With Objects](#)
- [Working With Links](#)
- [Working With MickL \(Model Writing language\)](#)
- [Tools](#)
- [Views](#)
- [Troubleshooting](#)
- [ICMS Tutorials](#)

Use the bookmarks to navigate through this guide – if your browser does not default to this view, toggle the navigation pane button to show and then click the bookmarks tab or choose window>show bookmarks.

We plan to update this reference as new or revised topics come to hand.

Updates will be distributed via our web site at

<http://www.cbr.clw.csiro.au/icms/> so make sure you visit regularly to ensure you have the most current version.

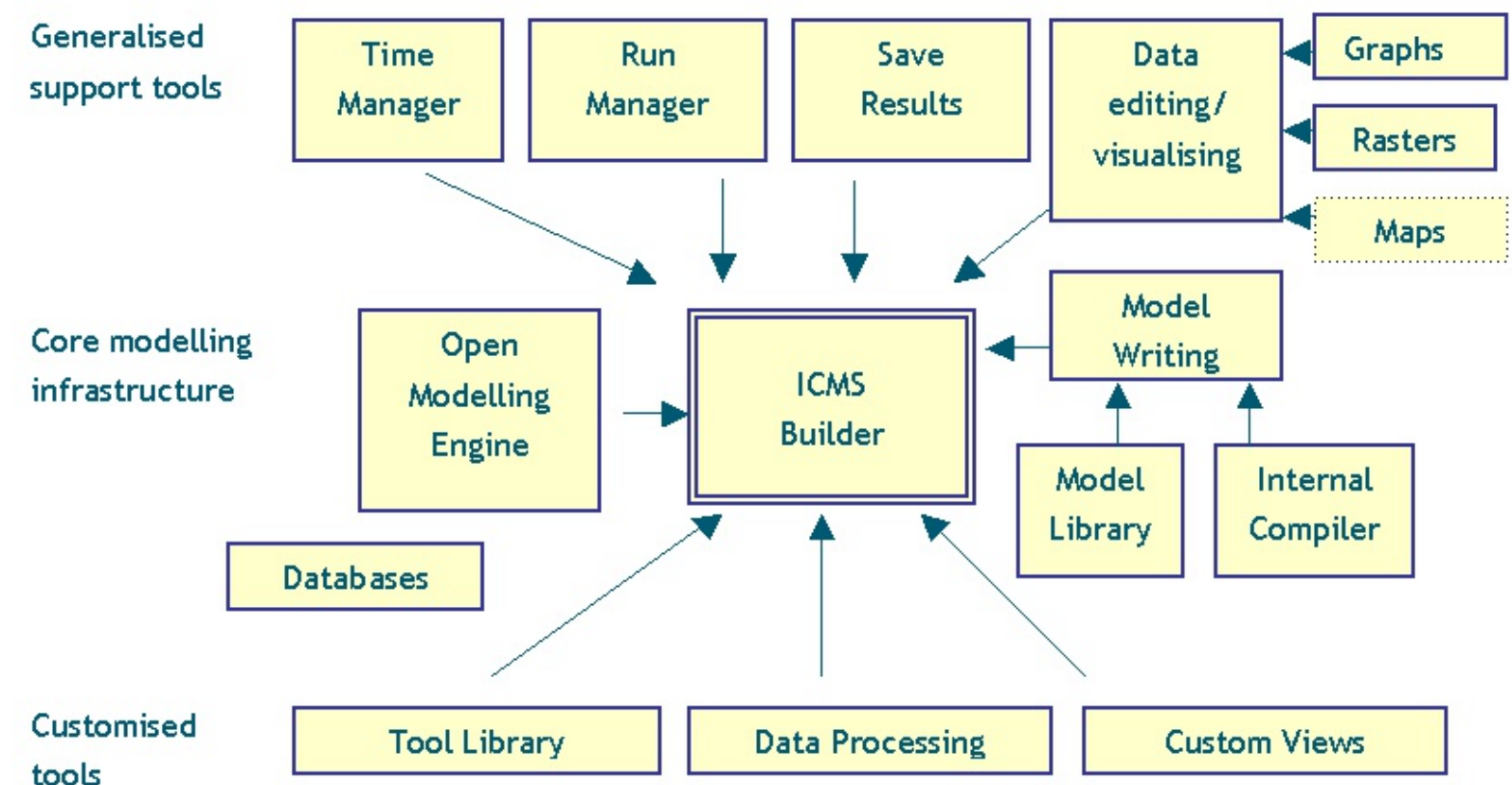
Architecture

This section introduces you to the components that make up the ICMS architectural framework and its internal language.

More:

- [ICMS Components](#)
- [Classes, Objects and Models](#)
- [The Open Modelling Engine](#)

ICMS Components



More:

- Time Manager
- Run Manager
- Save Results
- Data Editing / Visualising
- Open Modelling Engine
- Databases
- Model Writing
- Internal Compiler
- Model Libraries
- Tool Library
- Data Processing
- Custom Views

Time Manager

The time manager keeps track of temporal information in the system. It provides the run manager with the begin and end times of temporal data for running the models over a specified time period. Graph views use the time manager to label their axes when displaying temporal values. The temporal data view collects all the values in the system which have temporal attributes, and displays them in a Gantt chart, showing the spread of temporal data in the system.

Run Manager

The run manager defines how the models and system will operate when the user presses the Run button. It is required to manage a list of objects to run, to calculate the overlapping or full temporal run time (for temporal runs only) and to display the status of the model run. On completion of a model run it also writes output data and a run description to the databases so that those results may be used in future reports.

Save Results

Save Results gives the flexibility for a user-defined list of input and output variables to be stored for each run. These are stored in a Run Results library and can be retrieved later for comparison to other runs. This is a generalised implementation of a scenario comparison tool.

Data Editing / Visualising

Integral to the concept of useability for ICMS is data editing and viewing. For each type of data there are multiple editors available.

Graphs – Temporal or one-dimension data can be graphed in a customisable window. More data series may be added to the graph, or the style and colours may be configured.

Raster Maps – Two-dimensional data is best edited in a Paintbrush-like environment, using pens, brushes, magnifying glasses etc. The map view provides full editing of matrix data as well as customisable viewing of model results.

Vector Maps – When spatial data is fully integrated into ICMS there will be spatial views with which to display and edit vector-based information.

Open Modelling Engine

The OME maintains the databases and their connectivity. It provides an intelligent interface for doing the various tasks required by ICMS, such as adding objects, editing links, building classes, viewing data, and running the models.

Databases

The databases contain all the relational data being used by ICMS. Data is physically stored as streamed data within the one file, the .ICM file. During an ICMS session, data is held in memory until a Save commits the data to disk.

Model Writing

Models are written by users (or imported via model libraries). There are written in an internal C-type language (MickL) which is supported by many ICMS-specific functions. Models are associated with classes and objects.

Internal Compiler

To ensure speedy model code interpretation, ICMS has its own internal compiler. The compiler provides a basic level of debugging and compiles code to Intel machine code.

Model Libraries

Models can be imported and exported from ICMS using the model library. It keeps track of imported model groups, allowing the user to quickly add or remove whole families of models. This provides users the ability to distribute their models to other ICMS users, or to collect models from other users.

Tool Library

ICMS has a few general tools, such as flow statistics and flow direction algorithms. These are compiled as DLLs which can be registered into ICMS at any time. Any programmer can add to this library of tools.

Data Processing

ICMS provides some data processing and import/export routines. Other data processing routines can be added as tools which can be registered into ICMS at any time. Any programmer can add to this library of data processing tools.

Custom Views

Custom Views are tailored interfaces which sit onto of an ICMS 'project'. This is a rapid DSS building tool which is supported by a protocol with programming templates which demonstrate how these views can be built. Views are similar to tools in that they can be registered into ICMS at any time. Any programmer can add to the library of custom views.

Classes, Objects and Models

It is critical that you understand the concept of classes and objects.

Classes define the structure of the entities in the modelling domain. This includes data AND models and we define data by data templates, and models by model classes.

A class may define a general entity such as a catchment, or a process such as groundwater percolation. Then, **data templates** define the type of data which the modeller uses to describe the entity (eg in the case of a catchment, its name, area, land uses). Finally, **model classes** specify the kind of models which can be applied to the data to infer results, such as catchment assimilation models or rainfall-runoff models.

An **object** is an instance of a class. An object is created by assigning values to the data templates (for instance, assigning 12,000 to area for a particular catchment) and selecting the models we want to apply to these data sets. A simple example is:

- create a class called 'Subcatchment' with data templates 'Area' and 'Length of Stream'
- then create an object, ie an instance of class Subcatchment, called 'Gudjee Creek catchment', and enter '68' for 'Area' and '160' for 'Length of Stream'.

Models are described as symbolic equations. A model to simulate the Object's operation can be chosen from a library, or written from scratch, and links between the model's symbols (ie the model's input, state and output variables, and its parameters) and the Object's data instances (ie the data templates of class templates) are made to provide the model with real data. Building on the above example:

- create a model, LoadGeneration, with the symbols Area, LengthofStream and GeneratedLoad, within class Subcatchment
- associate this model with object 'Gudjee Creek Catchment'

Now, when the model runs for 'Gudjee Creek Catchment', the links between its symbols and the object's data are known and the correct data is passed to the model.

The Open Modelling Engine

This section describes the open modelling engine (OME) architecture, and how the use of classes and objects allows for model connectivity and model interchangeability.

The OME is a software architecture for model management. It is highly influenced by Systems Theory concepts. A model of a system is characterised by its input, state, and output variables, and by its parameters. Input, state and output variables are related by functional relationships which map the input variables to the outputs. This modelling paradigm is widely used and is particularly suited to represent environmental catchment processes. Many classes of models (functional, declarative, constraint-based) lend themselves to this structure. These definitions can be implemented using Object Technology to facilitate essential functions such as model prototyping, data access and model integration.

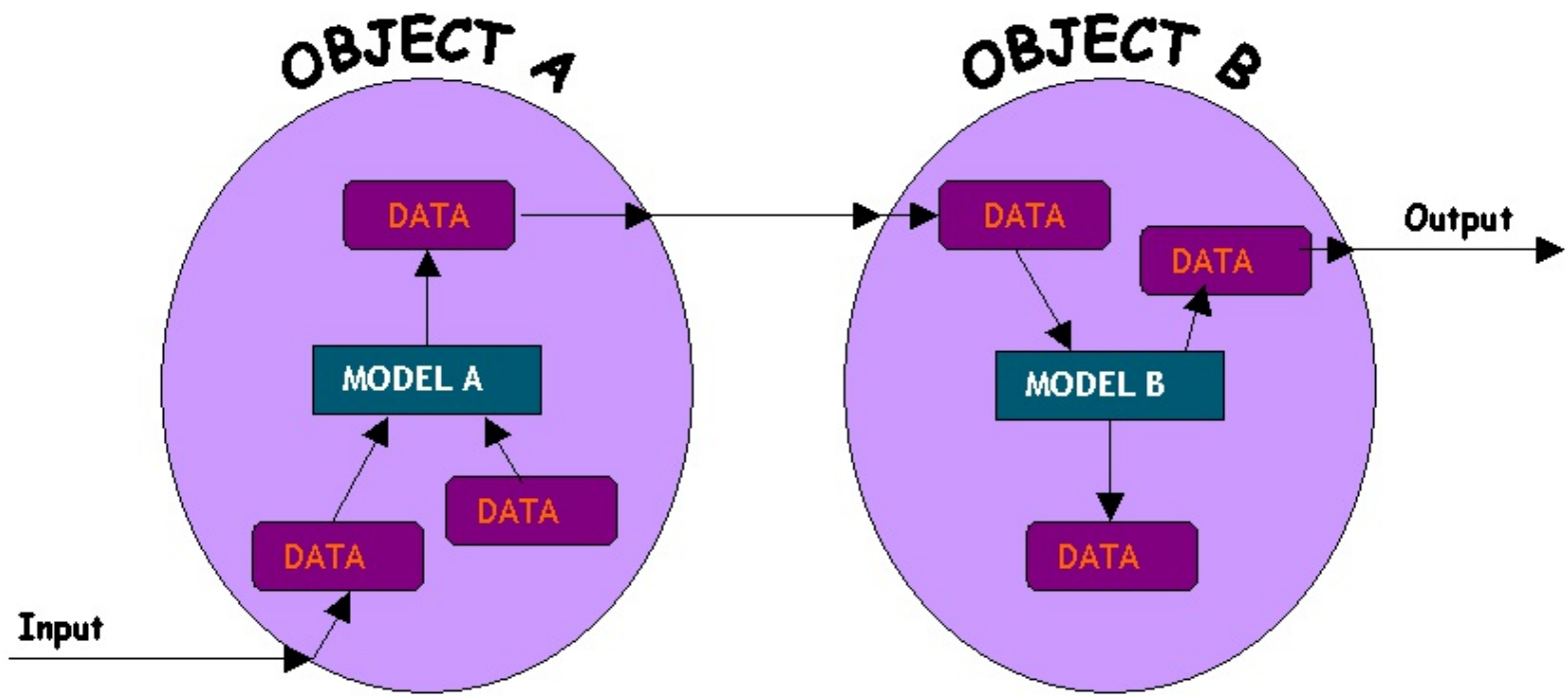
The structure of the OME is based on the concept of making models independent from data. For this purpose, we use Object Technology constructs such as classes and objects to construct a set of data structures apt to implement this logical separation.

Objects can be interconnected, creating complex modelling domains, using Object to Object links which are connections (ie links) between data instances in different domain objects. These links allow the exchange of information between objects.

Finally, the OME uses a Parser which is powerful interpreter which processes each Object by moving data along the data links and running the associated model. The results are saved in a database for future display or processing.

This approach to the design of the OME was taken after consideration of the requirements of the domain of interest, namely modelling water quality at catchment scale.

To recap, each Object contains data instances which describe the Object's parameters, inputs and outputs. These data are mapped via links to data instances in other Objects (see Figure). This structure allows the problem to be defined by using linked Objects to represent different parts of the catchment domain (e.g. streams, sources, sinks), and data flows along the links.



More:

- Model Connectivity
- Model Interchangeability

Model Connectivity

Connectivity between models is central to the OME. Objects may be linked to each other (parent/child relationship), data instances may be linked to each other (data transfer between Objects in a many-to-many relationship), and data instances may be linked to models (connecting the model to the Object). This connectivity allows the results of one model (associated with a Object) to be input into the next model.

A simple example would be two Objects which describe different sub-catchments. These sub-catchments are different in terms of landscape and transport processes. Different models (e.g. rainfall/runoff models) can be associated with each Object. The residual runoff from one Object can be passed as input to the next Object.

Model Interchangeability

One advantage of using the OME is the ability to interchange models. Different models may be linked to the same Object and associated data, allowing the modeller to begin with a simple model, and expand that model as more data become available. The earlier models can be stored away in a library, ready for reuse at any time. When a new model is developed it can be connected to its data using the model symbol to data template links. This allows different models to use the same data, and when a different model is chosen for a particular Object the associated links are restored.

ICMS Terminology

More:

- Child
- Classes
- Data templates
- Data Instances
- Functions
- .ICM
- ICMSBuilder
- Level
- Links
- Matrix
- MickL
- Models
- Objects
- OME
- Parent
- Parent/child link
- Project
- Scalar
- Sibling
- Sibling link
- Speed Menu
- Tools
- Views

Child

A child object is an object with a parent (and mentally sits at a lower level than the parent). Parents and children are merely a way of implementing hierarchy in design. Children are 'fully functioning' objects – they can have siblings, they can have their own children, and they can (and normally would) contain models. A child object can only have one parent.

Classes

Templates from which to build objects. Classes contain data templates and models.

Data templates

Descriptions of the structure of data instances within objects. Data templates are described within classes.

Data Instances

An instance of a data template within an object. Data instances contain real data (either entered by the user or calculated by a model).

Functions

Are routines within the MickL language. A large range of mathematical, temporal and data management functions are provided.

.ICM

The project file. Each project has one .ICM file, and an associated .DSK file which is a copy of the state of the desktop last time the project was saved in ICMS.

ICMSBuilder

The engine of ICMS. This is the name of the executable icmsbuilder.exe which is opened each time you start ICMS.

Level

A term to describe the ability to 'stack' objects to suit your problem definition. An object can have many children who then sit at the level 'below' it (it is now a parent). The number of levels is only limited by the ability to dissect your problem definition and solution design.

Links

Connections between objects. Links are established by connecting data instances from different objects.

Matrix

ICMS stores numeric data in 'values'. Each value can be either a scalar or a two-dimensional matrix. They are represented as [width, height]. This is the reverse to the mathematical or engineering representation, which is [rows, columns]. For example, getting the height of a matrix will return the number of rows. When creating a matrix, the width is first specified and then the height. Matrix cells can be of byte, integer, single or double precision.

MickL

The language used to write model formulae. It is very C like in its command and program control syntax and has an extensive range of ICMS-specific functions.

Models

Mathematical equations whose variables are classified as output, input or local variables. Models run inside objects and the model's variables (which are merely symbols) are connected to the object's data instances to provide the model with real data.

Objects

'Things' which contain data instances and a model. Objects are created from a class. Objects can be linked to other objects in a sibling relationship or in a parent-child relationship.

OME

Open Modelling Engine – the internal data and model manager.

Parent

A parent object is an object with children (which sit 'below' the parent). A parent cannot contain models.

Parent/child link

A link between objects at different levels, but only one level apart.

Project

ICMS stores each system (containing a system view, classes, objects, models, data and results) in one file, called a project file (.ICM). Projects are totally self-contained and can be transferred from ICMS user to ICMS user simply by copying the .ICM file.

Scalar

ICMS stores numeric data in 'values'. Each value can be either a scalar or a two-dimensional matrix. A scalar can be of byte, integer, single or double precision.

Sibling

Objects are siblings if they have the same parent. If there is only one level, the (virtual) parent is the ICMS Root Object.

Sibling link

A link between objects on the same level (not between parent/child).

Speed Menu

Accessible by clicking the right mouse button when the mouse is located within a view. Each view has an associated speed menu. Entries in the speed menu are also available from the Main Menu.

Tools

A set of routines which can be manually invoked by a user (not by a model) from a menu. They are commonly data processing tools. These are written and compiled as DLLs by experienced Delphi programmers.

Views

Tailored interfaces closely coupled with projects. These are written and compiled as DLLs by experienced Delphi programmers.

Working With Classes

It is critical that you understand the concept of classes and objects. Class templates define the structure of the entities in the modelling domain. This includes data AND models and we define:

- data by data templates, and
- models by model classes.

A class may define a general entity such as a catchment. Then, **data templates** define the type of data which a modeller uses to describe the entity (eg the total catchment area, which must be a real number, expressed in square kilometres). Finally, **model classes** specify the kind of models which can be applied to the data to infer results, such as catchment assimilation models or rainfall-runoff models.

A **object** is an instance of a class (within the architecture these are called Objects). An object is created by assigning values to the data templates (for instance, saying that the total catchment area is 12,000 sq km) and selecting the models we want to apply to these data sets. A simple example is:

- you create a class called Subcatchment with data templates area and length of stream
- you then create an object, ie an instance of class Subcatchment, called 'Gudjee Creek catchment', and enter '68' sq km for the area and '160' km for the length.

Models are described as symbolic equations. A model to simulate the Object's operation can be chosen from a library, or written from scratch, and links between the model's symbols (ie the model's input, state and output variables, and its parameters) and the Object's data instances (ie the data templates of classes) are made to provide the model with real data. A simple example is:

- You create a model, LoadGeneration, with the symbols Area, LengthofStream and GeneratedLoad, which is linked to class Subcatchment (and thus available to its objects).
- You link this model to 'Gudjee Creek Catchment' (a Subcatchment object).
- When the model runs for 'Gudjee Creek Catchment', the links between its symbols and the object's data are known and the correct data is passed to

the model.

Objects can be interconnected, creating complex modelling domains, using Object to Object links which are connections (ie links) between data instances in different objects. These links allow the exchange of information between objects.

Finally, the OME uses a Parser which is powerful interpreter which processes each Object by moving data along the data links and running the associated model. The results are saved in a database for future display or processing.

This approach to the design of the OME was taken after consideration of the requirements of the domain of interest, namely modelling water quality at catchment scale.

To recap, each Object contains data instances which describe the Object's parameters, inputs and outputs. These data are mapped via links to data instances in other Objects (see Figure 1). This structure allows the problem to be defined by using linked Objects to represent different parts of the catchment domain (e.g. streams, sources, sinks), and data flows along the links.

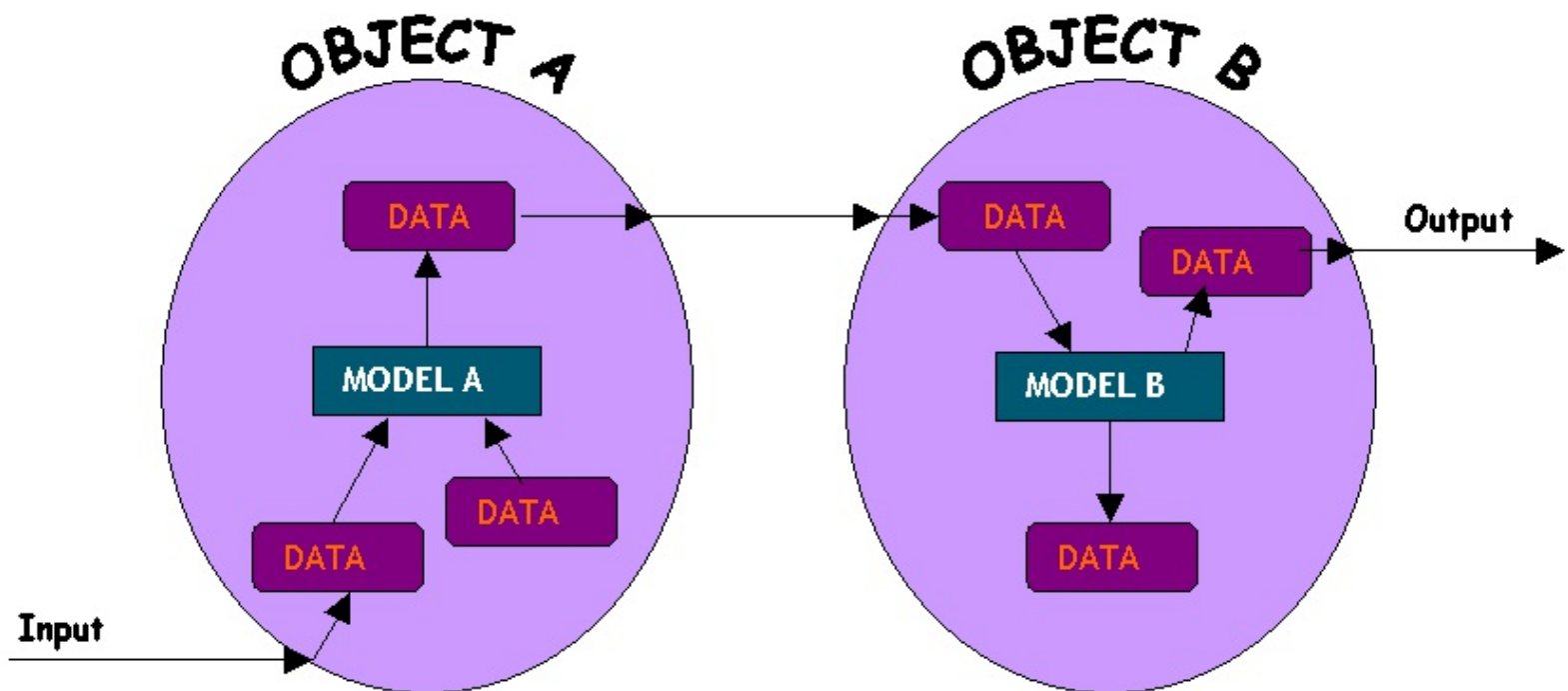


Figure 1, Objects showing object, data, model and link relationships

More:

- [Model Connectivity](#)
- [Model Interchangeability](#)
- [What are Classes?](#)
- [Adding Classes](#)

- Viewing Classes
- Special ICMS Classes


Model Connectivity

Connectivity between models is central to the OME. Objects may be linked to each other (parent/child relationship), data instances may be linked to each other (data transfer between Objects in a many-to-many relationship), and data instances may be linked to models (connecting the model to the Object). This connectivity allows the results of one model (associated with a Object) to be input into the next model.

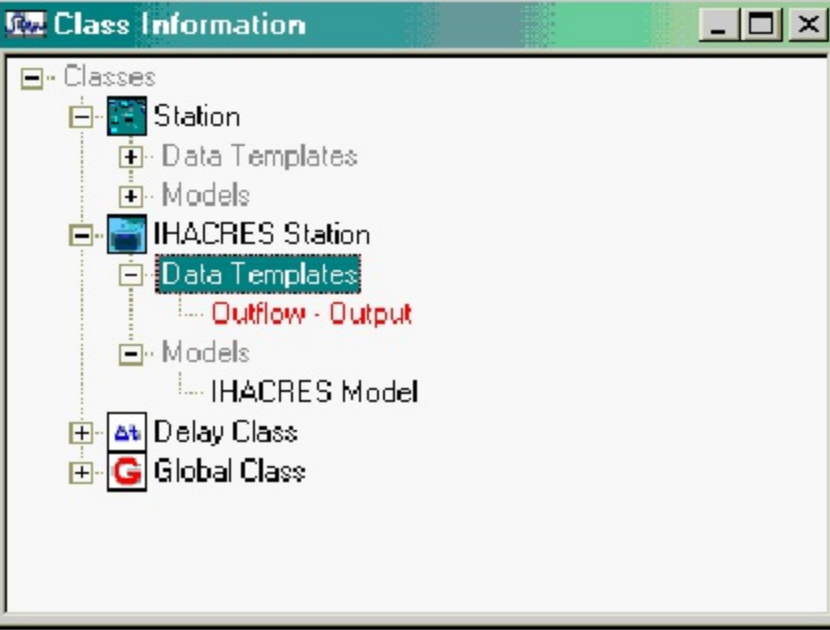
A simple example would be two Objects which describe different sub-catchments. These sub-catchments are different in terms of landscape and transport processes. Different models (e.g. rainfall/runoff models) can be associated with each Object. The residual runoff from one Object can be passed as input to the next Object.

Model Interchangeability

One advantage of using the OME is the ability to interchange models. Different models may be linked to the same Object and associated data, allowing the modeller to begin with a simple model, and expand that model as more data become available. The earlier models can be stored away in a library, ready for reuse at any time. When a new model is developed it can be connected to its data using the model symbol to data template links. This allows different models to use the same data, and when a different model is chosen for a particular Object the associated links are restored.

Click on the  icon on the toolbar. This opens the Class Information View and adds the Class menu to the main Menu. Click on Class and the class menu list will appear.





What are Classes?

Classes define the structure of the entities in the modelling domain. This includes data AND models and we define:

- data by data templates, and
- models by **model** classes.

A class may define a general entity such as a gauging station. Then, **data templates** define the type of data which a modeller uses to describe the entity (eg name, staging details, flow). Finally, **models** specify the kind of models which can be applied to the data to infer results, such as flow routing or rainfall-runoff models.

In this figure, the IHACRES Station class is expanded to show that it contains one data template - Outflow which is of data type Output) and one model – the IHACRES Model.

Classes are a simple and powerful way to build the 'meta-data' behind a system. While you may (and probably will) have multiple objects of a class (eg many subcatchments of class Subcatchment), you need only describe the data and the model(s) once.

To recap,

Classes are where you define and edit DATA TEMPLATES

Classes are where you define and edit MODELS

Classes DO NOT CONTAIN DATA

Classes DO NOT RUN MODELS.

Data is stored within OBJECTS


Models are run within OBJECTS.

Adding Classes

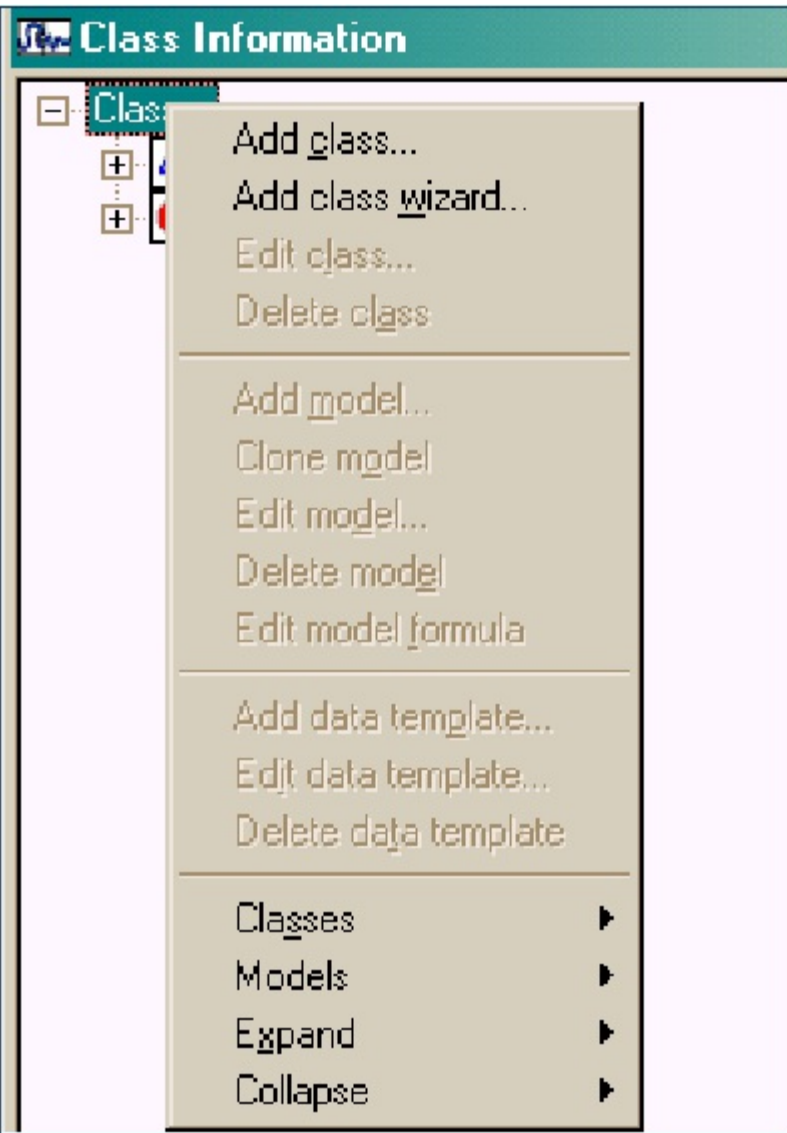
When you start a NEW Project, ICMS automatically opens the Class Information View.



At this stage only the ICMS Delay and Global classes (described later in this section) exist.

If you are working with an existing Project, click on the  icon on the toolbar. This will display the Class Information View. Then you can work from this View's speed menu or from the Classes menu. The following details assume that you are working with the speed menu.

Select Classes and press the right hand mouse button. The speed menu appears.



Add class opens the Class Properties dialog box. This lets you give the class a Name, a Description and an Icon.

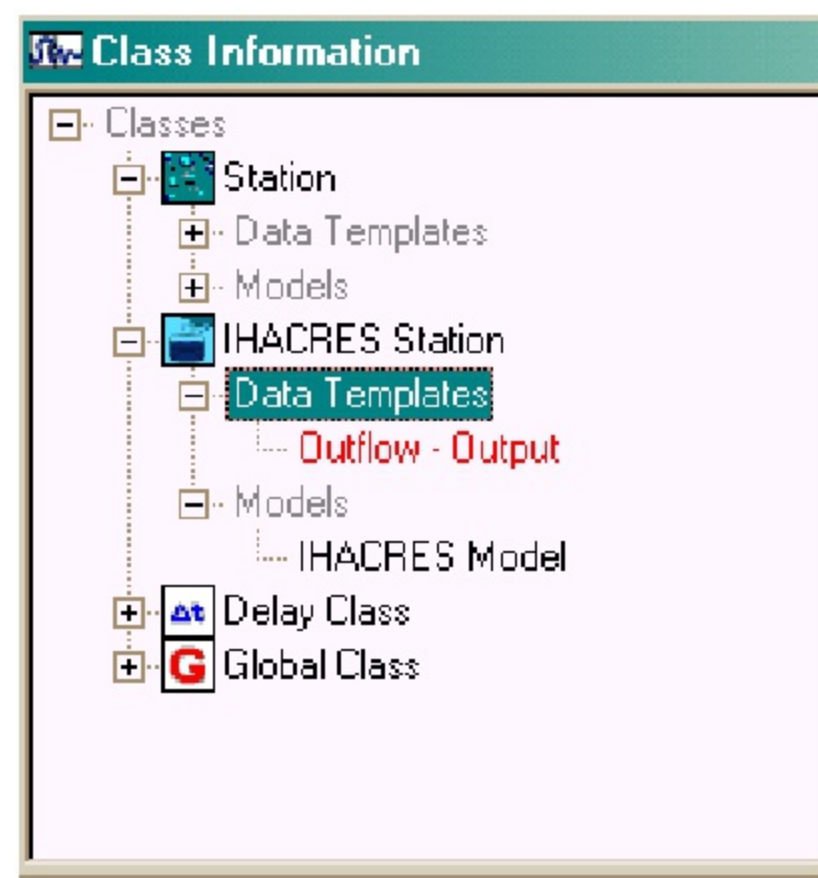
Add class wizard is more powerful and lets you add not just the class properties, but also the class's data templates and models.

ICMS provides a directory of icons (stored in the ICMS vxx\Icons subdirectory). You can also create your own. To be consistent with other icons, they should be 20x20 pixels. You can change the icon associated with a class at any time. You can add/edit data templates and models at any time.

When you select an existing class, the model and data template menu items become active.

Viewing Classes


The Class Information view contains the class and template structure of the ICMS project. At the root of all templates is the Class node. Under this are listed all the classes defined so far with their icon and name shown. Each class contains Data Templates and Models. The data templates are descriptions of the data available within the class, usually used by one or more of the models. The models are algorithms which process the data. Under each model there is a list of the symbols (variables) used within the model, and those which are linked to data templates are shown.



As this is a tree structure you can open and close any branch of the tree by clicking on the + or - boxes.

The right mouse button is used to access many functions in this view by right-clicking on a particular node in the tree. Only those menu options which are valid for the particular node will be enabled. These functions are also available through the Classes menu.

This view is accessed from:

- the  icon on the toolbar
- the View|Class Information menu item.

Special ICMS Classes

More:

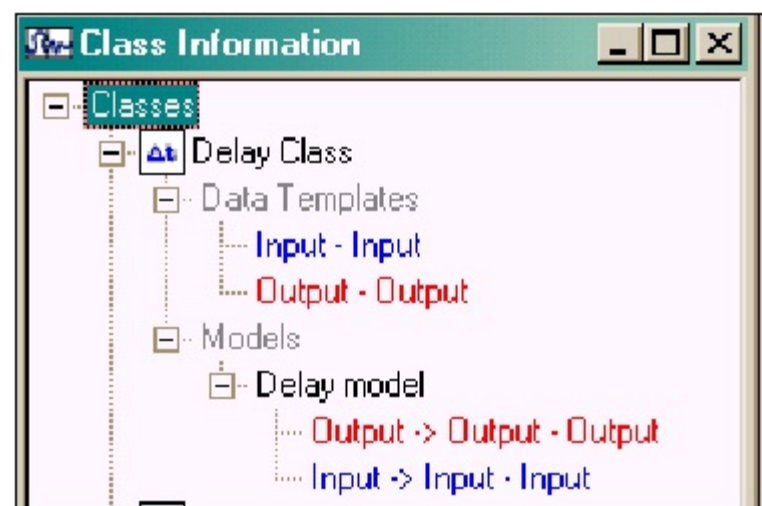
- [The Delay Class](#)
- [The Global Class](#)

The Delay Class

This class is a delta object and is used to simulate the delay of one or more time steps, rather than coding delays into the model. The class has 2 data templates and one model, the Delay model. This model has one line of code, namely:

```
Output = Input;
```

Objects of this class are placed between objects where the input of one object is the output of another.



The Global Class

You cannot edit the global class – it is managed by ICMS. It is used to record the list of objects in the System View (which are children of the ICMS root object); and to store global variables (underscore 1st character). This figure demonstrates an example of the use of this class. In this example, year is common to all objects of a particular class. Rather than store the same value for year in each object, it is stored only once within a global object. ICMS creates the global object (from the Global Class) as soon as a global variable is created within a model.



Working With Models

Models are associated with classes. One class can contain many models. However, when an object is created, it can have only one (or no) model associated with it.

More:

- [Model Connectivity](#)
- [Model Interchangeability](#)

Model Connectivity

Connectivity between models is central to ICMS. Objects may be linked to each other (parent/child relationship), data instances may be linked to each other (data transfer between Objects in a many-to-many relationship), and data instances may be linked to models (connecting the model to the Object). This connectivity allows the results of one model (associated with a Object) to be input into the next model.

A simple example would be two Objects which describe different sub-catchments. These sub-catchments are different in terms of landscape and transport processes. Different models (e.g. rainfall/runoff models) can be associated with each Object. The residual runoff from one Object can be passed as input to the next Object.


Model Interchangeability

One advantage of using ICMS is the ability to interchange models. Different models may be linked to the same Object and associated data, allowing the modeller to begin with a simple model, and expand that model as more data become available. The earlier models can be stored away in a library, ready for reuse at any time. When a new model is developed it can be connected to its data using the model symbol to data template links. This allows different models to use the same data, and when a different model is chosen for a particular Object the associated links are restored.

Working With Objects



Click on

the  icon on the toolbar. This opens the Object Information View and adds the Object menu to the main Menu. Click on Object and the object menu list will appear.

Object	Run	View	Tool
Add class...			
Edit class...			
Delete class			
Add object...			
Edit object...			
Delete object...			
Clone object			
Show data tree			
Show object system			
Edit all scalar values			
Edit model formula			
✓ Enabled in run			
Delete saved data			
Value			▶
Add data link...			
Edit data link...			
Delete data link			
Classes			▶
Objects			▶
Expand			▶
Collapse			▶

More:

■ [What are Objects?](#)

- Adding Objects
- Viewing Objects
- Object Properties
- Cloning Objects
- Show Data Tree
- Show Object System
- Edit all Scalar Values
- Edit Model Formula
- Enabled in Run
- Delete Saved Data
- Values

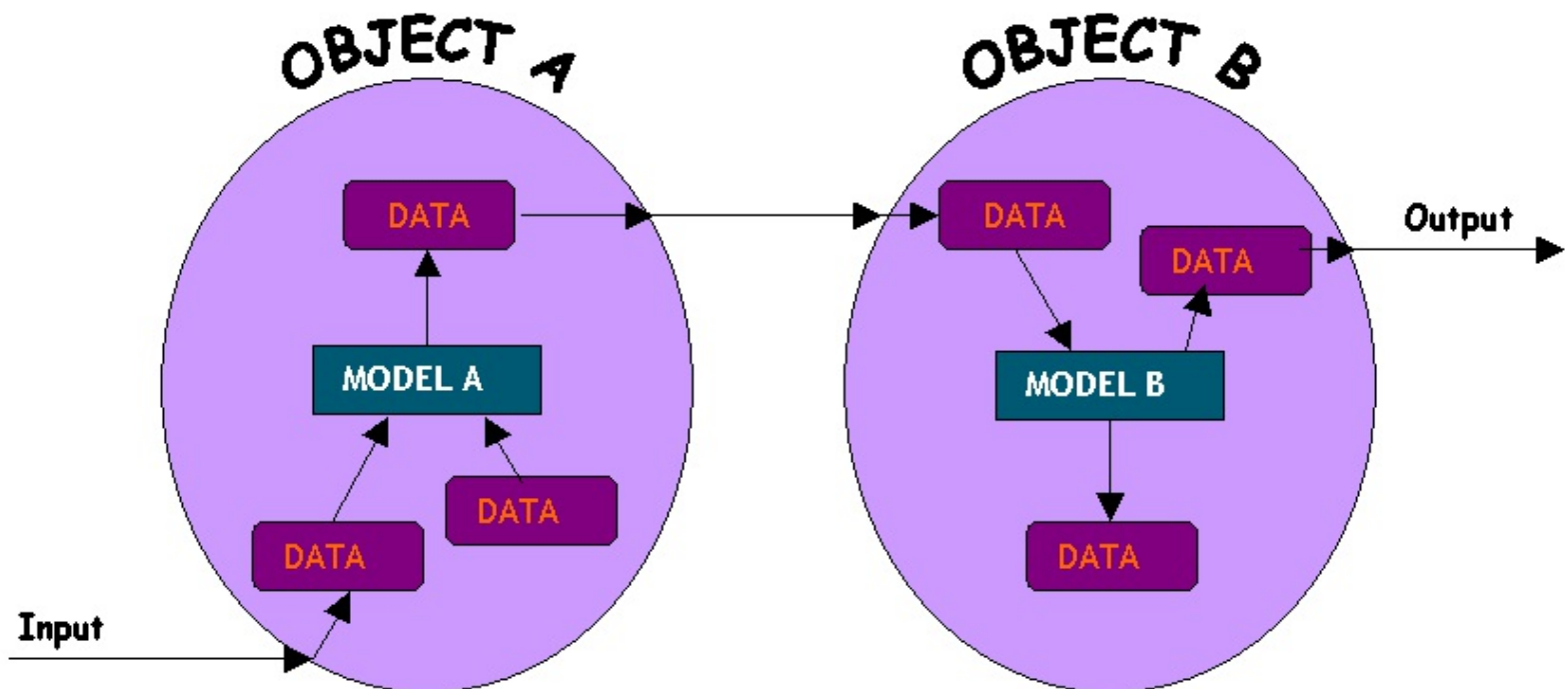
What are Objects?

Objects are 'instantiations' of classes. They are core to the ICMS way of building representations of your project. You create objects which describe processes or locations. An object is 'real', whereas a class is just a description. An object contains data, whereas a class simply describes the general form of that data (ie its template). An object usually contains a model (unless it is a parent object).

Objects can be interconnected, creating complex modelling projects, using Object to Object links which are connections (ie links) between data instances in different objects. These links allow the exchange of information between objects.

Finally, ICMS has a Run Manager which processes each Object by moving data along the data links and running the associated models. The results are saved and accessed by querying the objects.

To recap, each Object contains data instances which describe its parameters, inputs and outputs. These data are mapped via links to data in other Objects (see Figure). This structure allows your problem to be defined by using linked Objects to represent different parts of the catchment domain (e.g. streams, sources, sinks), and data flows along the links.




Adding Objects

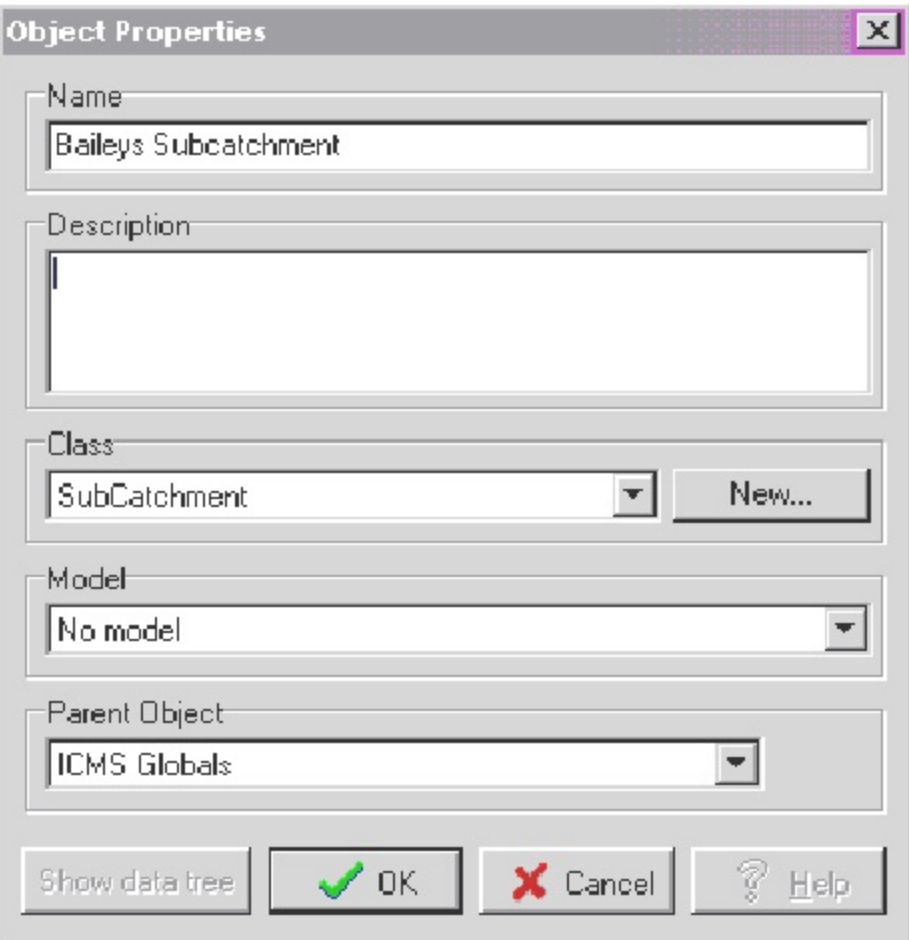
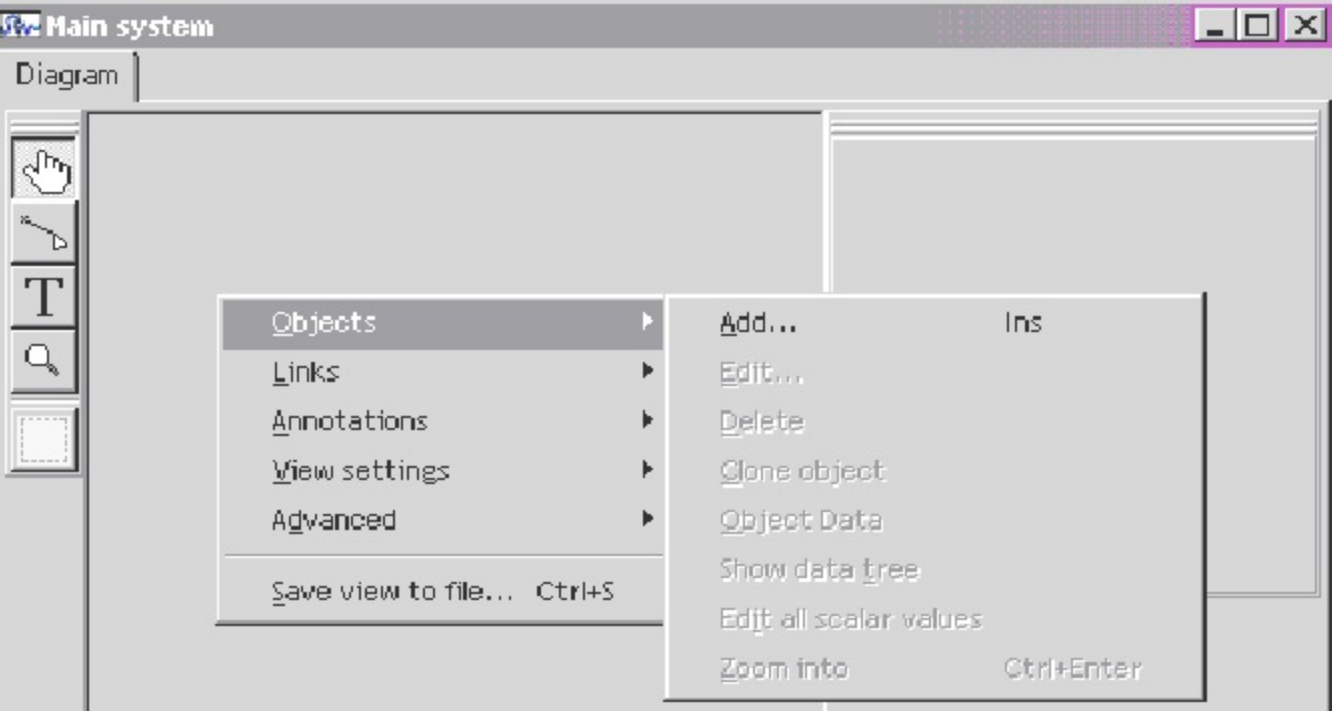
Objects are most easily added via the speed menu (right click anywhere within the panel) of the **System View**. This spatially locates them on the panel. Objects are then described using the Object Properties dialog box, which can be accessed from several places in ICMS, including:

- System View window – select mode, select an object, double click
- System View window – select mode, right click, select Objects from the speed menu
- Object Information window – select an object, double click
- Object Information window – select an object, valid Object edit activities available on the speed menu

The Data Properties dialog box appears, regardless of your mode of entry. Of course, the options that are available on the active window are also available by selecting the appropriate menu item on the top menu bar.

To work through an example:


1. You must have created the object's class first.
2. Go to the System View by clicking  on the task bar.
3. Press the right mouse button to display the System View speed menu and select the Objects|Add.
4. Fill out the Object Properties box that appears. Click OK.
5. You have now created an object of your chosen Class.

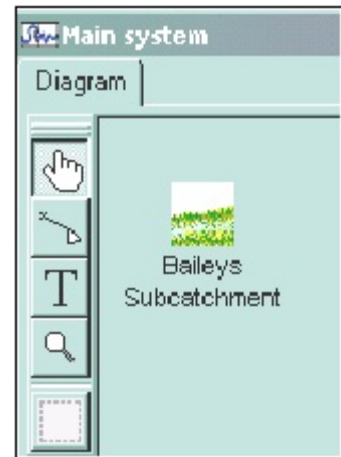


Viewing Objects

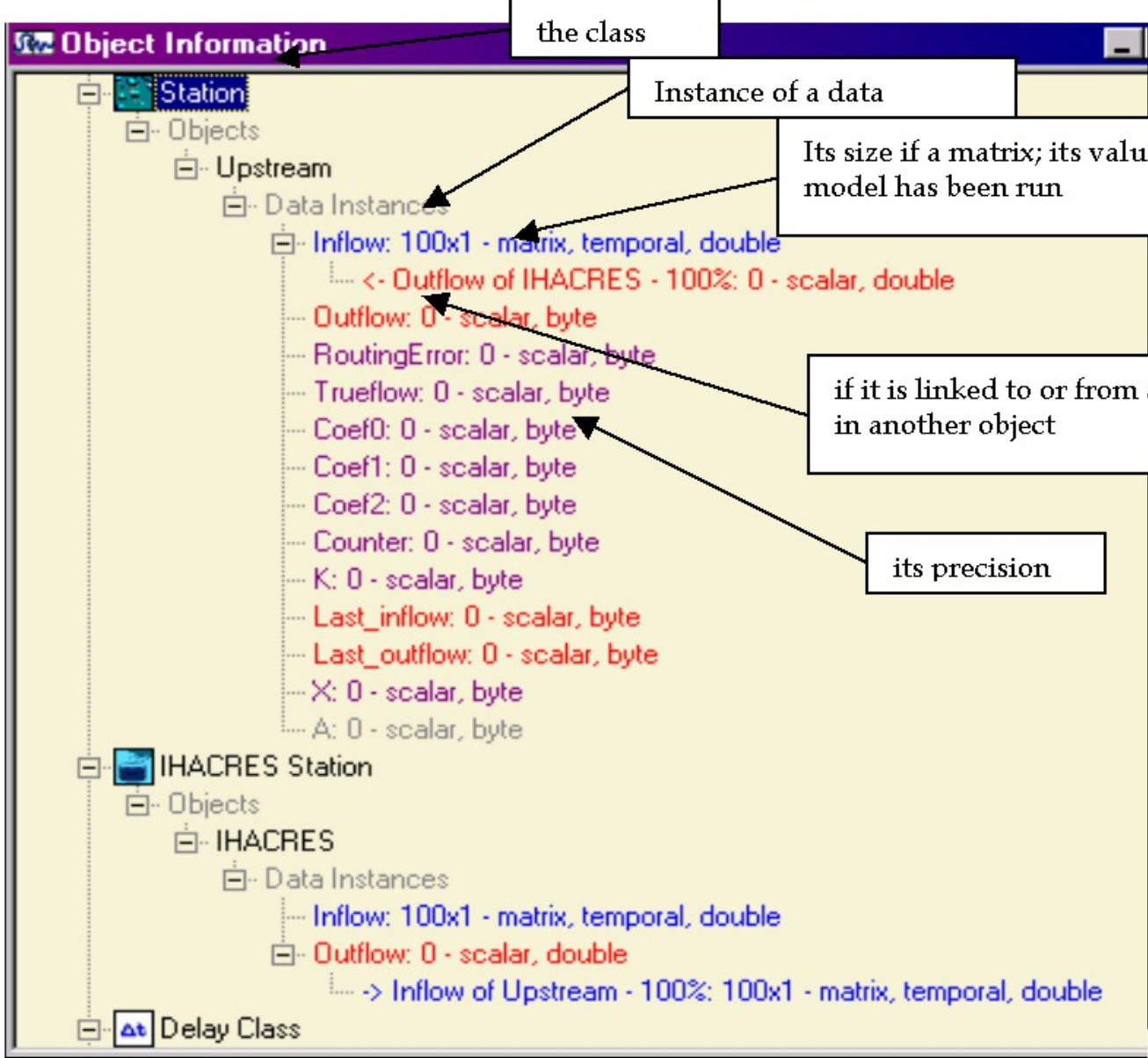
ICMS provides 2 views of objects – a tree view and a visual view.

The visual view is available via

- View|System on the main menu
- the  tool.



The tree view is accessed via View|Object Information on the main menu and the  tool.



the class

Instance of a data

Its size if a matrix; its value if a scalar and a model has been run

if it is linked to or from a data instance in another object

its precision

Object Properties

There are 5 properties of an object that need to be established. These are:

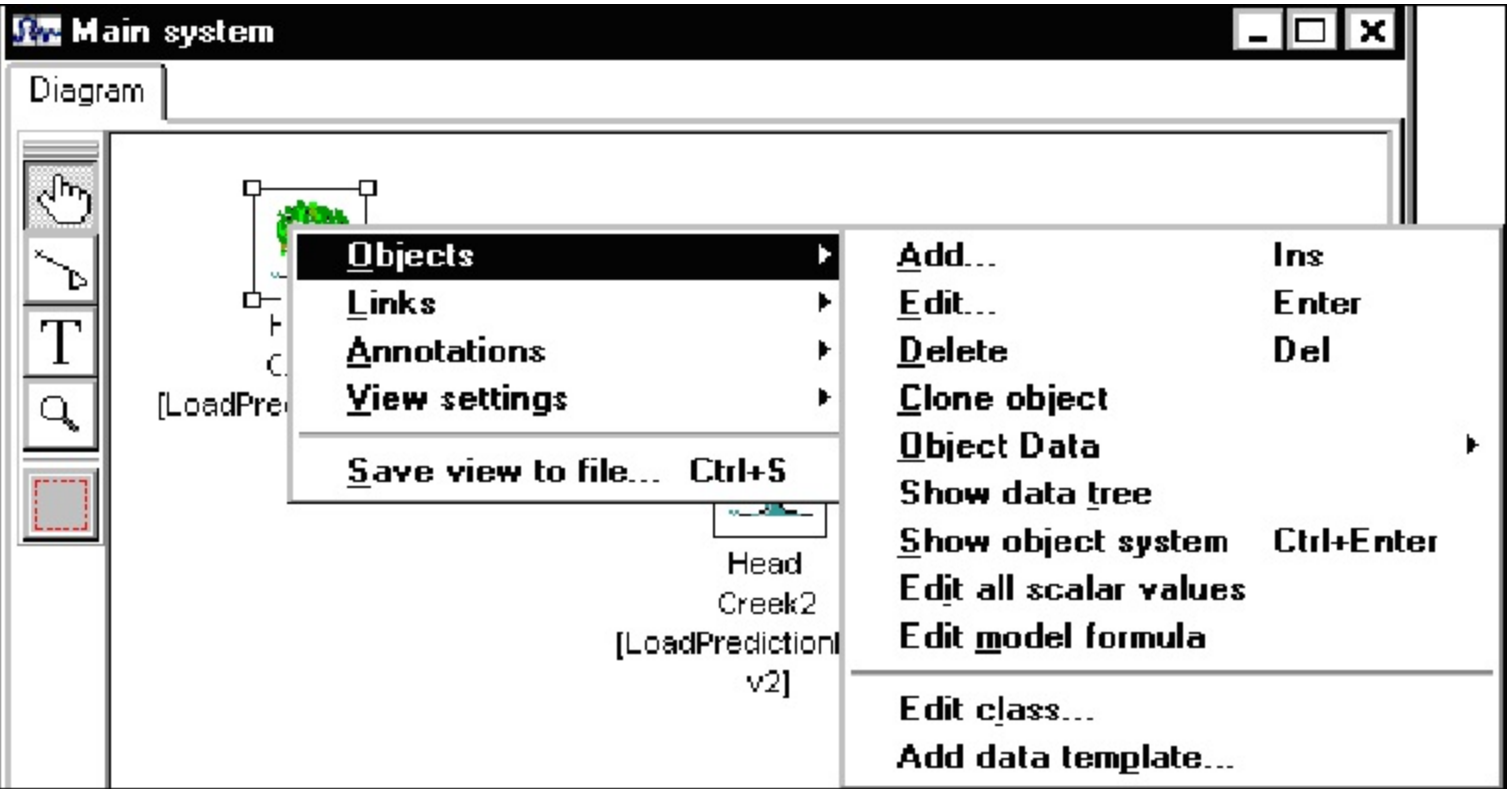
- its name (mandatory)
- its class (mandatory)
- its model (if its class has models). If the class has multiple models, then different models can be selected. However, an object can use only one model at a time.
- its parent object (default is ICMS)
- a description of the object (optional)

An object also contains data. Each object has data instances of any or all of the data templates defined for its class. The data instances associated with an object are usually defined by the model that it contains. The data is 'loaded' manually by the user, received from a linked source object, or calculated by the model.

Cloning Objects

Objects can be copied by cloning. Cloning copies an object's data and associated model. If the object has children, it also copies the children and the linkages.

Use the speed menu on the System View (right click, Objects Clone object)



Or use the speed menu on the Object Information View (right click on the object).

ICMS has a naming convention when cloning, adding a number to the end. Thus, if your object were called Creek 1, the 1st copy would be called Creek 12, the 2nd copy would be called Creek 13. If you then copied Creek 13, its 1st copy would be called Creek 132.

Show Data Tree

Opens a special case of the Object Information View showing only the data for the object.

Show Object System

This is the mechanism whereby you open another System View to draw the children of an object. If the object has no children, the view will be empty.

Edit all Scalar Values

This is a data editing tool to facility more rapid data entry. It opens one edit dialog box for all the scalars in an object. This circumvents needing to open individual numeric views for each data instance.

Edit scalar values for - Head Creek

1

TotalSubcatchmentLoad (double)

0

InLoad (double)

0

InLandUseLoads (double)


OK

Cancel

Edit Model Formula

Once a system is set up, it is often easier to access the model formula via an object containing that model, rather than its class. Remember that editing the model changes the model for all objects that use the model, not just the one from which you have accessed the model code.

Enabled in Run

ICMS provides the ability to enable and disable objects during a run. You can do this by toggling this menu item, or via the  icon on the System View.

Delete Saved Data

ICMS provides a facility whereby data generated during a run can be stored (in the Run Library). Any results written to this library are called Saved Data and are associated with an object. This menu item is simply a convenient way to delete those data.

Values

Values gives you access to a range of ways of adding, editing, viewing and exporting each data instance.

View value uses the default view.

Add temporal information is only relevant to a temporal matrix.

View value

View value as a raster

View value as numeric

View value as a graph

Import...

Import wizard...

Export...

Copy **Ctrl+C**

Paste **Ctrl+V**

Add temporal information...

Working With Links

Links allow the exchange of information between objects.

Links are easily established between sibling links. There are no restrictions on the number of links that can be established between the same two objects and there are no restrictions on the number of other objects than an object can be linked to. There are restrictions on establishing from and to links between the same objects as this can lead to circularity, and thus be invalid.

Links between parents and children have stricter rules:

- parent/child links can only go up or down one level, ie no parent/grandchild relationships are allowed. Of course, you can have a parent/child/grandchild relationship.
- a parent cannot contain a model. (With this restriction, a child can link to and from a parent. Without this restriction, ie if there were a model in the parent, the OME would determine that circularity could exist and declare the link invalid.)
- a child can have only one parent (is this a social comment?).

More:

■ [Link Properties](#)

■ [Establishing Links](#)

Link Properties

There are 4 properties of a link that need to be established. These are:

- from (source) object
- to (destination) object
- percentage of data that 'travels' along the link. Usually this is 100% unless you are using this feature to disaggregate data in some way
- description of the link.

These are entered/edited within the Links dialog box, which can be accessed from many places in ICMS, including:

- System view window – link mode, rubberband from one object to another
- System View window – select mode, double click on a link
- System View window – select mode, select an object, right click, select Links from the speed menu
- System View window – select mode, select a link, right click, select Links from the speed menu
- Object Information window – select an object or one of its data instances, right click, select Links from the speed menu
- Object Information window – select an object and drag and drop to another object.

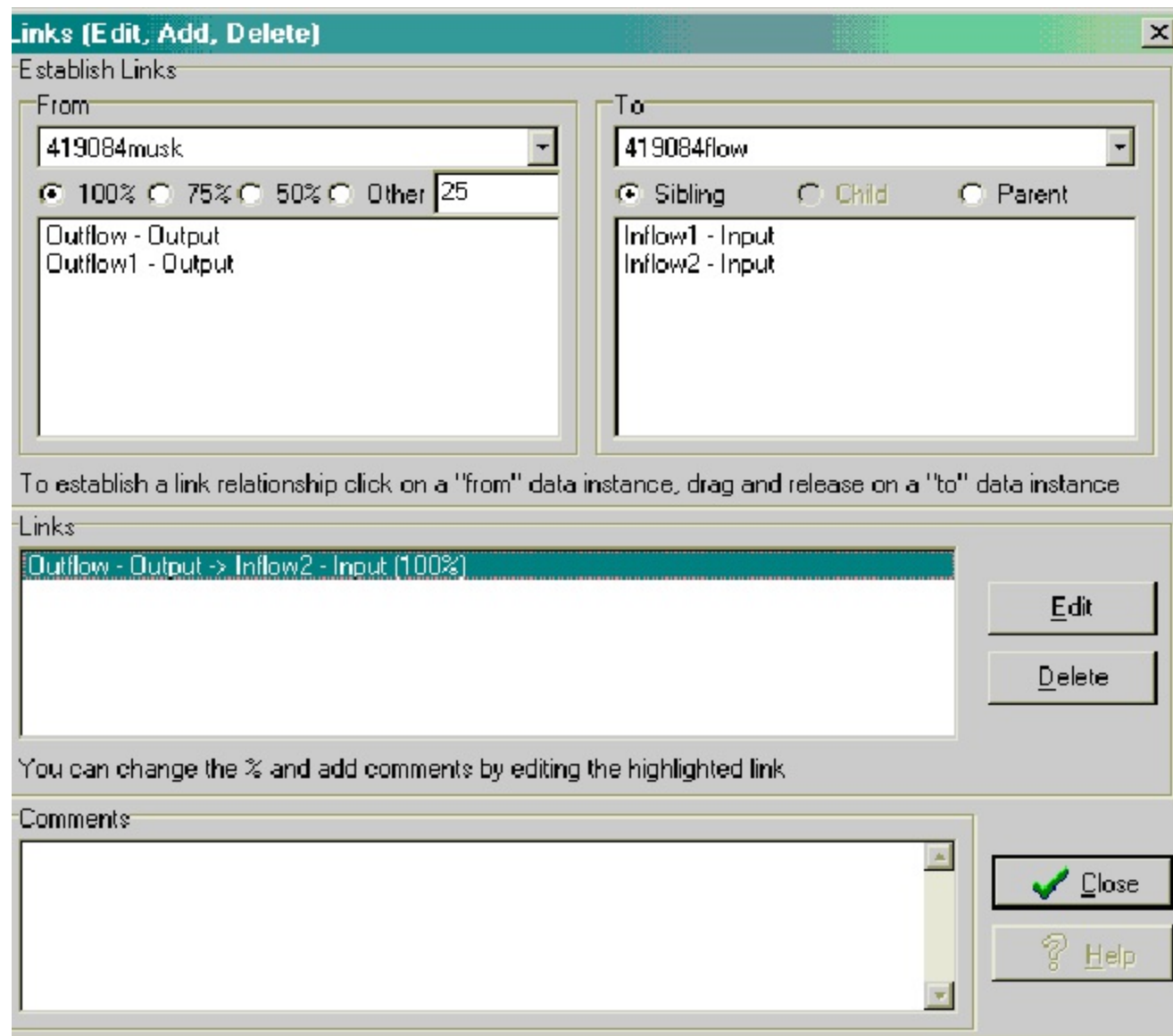
Of course, the options that are available on the active window are also available by selecting the appropriate menu item on the top menu bar.

The Links dialog box appears, regardless of your mode of entry. However, the mode of entry influences the content of the Links dialog box because ICMS uses its knowledge to restrict the links that you can create, depending on existence of objects and their relationships to each other.

More:

■ [The Links Dialog Box](#)

The Links Dialog Box



There are 4 discrete components of this dialog box:

- From – list of 'from' objects (list of objects and, for each selected object, its output data instances)
- To – list of 'to' objects (list of possible to objects and, for each selected object, its input data instances)
- Links - list which builds up as you create links
- Comments - box to describe each link.

Associated with the Destination object are 3 radio buttons:

- sibling
- child
- parent

parent.

ICMS uses its knowledge of objects and their relationships to help you navigate your way through links. (It also helps ICMS enforce rules such as children can't link to more than one parent because ICMS won't make invalid options available).

If the Source object has sibling objects, then the sibling radio button will be enabled and a list of sibling objects will appear in the right hand box. By default the data instances of the 1st sibling object will be displayed.

If the source object has children, then the child radio button will be enabled and a list of child objects will appear in the right hand box. By default the data instances of the 1st child object will be displayed.

If the source object has a parent, then the parent button will be enabled and the parent and its data instances will be displayed.

Of course, all 3 buttons or any combination of them can be enabled for a particular object.


Once in the Links dialog box, you have total flexibility to create valid links between all objects in the project, regardless of your entry point, simply by selecting a different source object in the left hand box.

Establishing Links


More:

-  [Sibling Links](#)
-  [Parent-Child Links](#)

Sibling Links

Sibling links are most easily established on the **System View** because you can graphically draw links between objects in the same System View window, using the  button. Then double clicking on the link brings up the Link dialog box.

Parent-Child Links

It is not possible to use the  button to add links between parents and children because they reside on different System View windows. Use **Zoom Into** to create a child system view for an object. Create some objects on this System View. Then linking can be done by selecting the parent (or child) object and bringing up the Link Dialog box.

Working With MickL (Model Writing language)

MickL is the ICMS internal language. It contains an extensive range of mathematical and program control functions. Many of these functions are purpose-built to provide powerful model processing ability, especially of matrices and temporal data. Remember that ICMS stores numeric data in "values". Each value can be either a scalar or a 2D matrix with precision of byte, integer, single or double. A full list of functions and MickL syntax is available later in this reference section.

More:

- [Model Template](#)
- [MickL Syntax](#)
- [Data Templates and Model Symbol Names](#)
- [Data Types](#)
- [Resolution](#)
- [Statements](#)
- [Functions](#)
- [Program Control Structures](#)
- [MATHEMATICAL OPERATIONS](#)
- [MATRIX AND SCALAR FUNCTIONS](#)
- [GRID FUNCTIONS](#)
- [TIME FUNCTIONS](#)
- [TOOLS FUNCTIONS](#)

Model Template

As soon as a model is defined (by typing in a model name when describing a class), the following template is created into which you can start adding your own code

```
function Initialisation()  
{  
    // put your initialising routines here  
    return 0;  
}  
function Main()  
{  
    // put your main program here  
    return 0;  
}  
function Finalisation()  
{  
    // put your finalising routines here  
    return 0;  
}
```

The template contains three predefined functions which require no parameters and return no values:

Initialisation() is run before the first time step.

Main() is run at each time step of the model run.

Finalisation() is run after the last time step.

MickL Syntax

The MickL language is a subset of C. The syntax is identical to C, except for:

- No variable declarations – variable types are not required in ICMS since the data type of a variable is dependent on how it is used outside of the model.
- Case of variable names is important – variables with an underscore 1st character are global to the system. Variables with lowercase 1st character will be used locally, ie only within the scope of the function they are used in, and are not persistent after a model run. Variables with an uppercase 1st character are persistent between model runs, ie they are stored in a database and can be edited and viewed outside the model.
- No passing of global variables – since variables with uppercase 1st character are considered global to the model, they do not need to be passed into a function. The compiler will therefore prevent a function declaration which has a global variable as a reference or value parameter.
- No nesting of functions – functions can not be declared within another function..
- No recursive functions – a function may not be called within itself. This is due to the difficulty associated with managing stacks in ICMS, hence the feature has been disabled.

Data Templates and Model Symbol Names

When a model is compiled, ICMS links the model symbol names (ie the variable/parameter names that you have used in the model) to data templates. Data template names can be long and contain spaces etc. Model symbol names cannot contain spaces or characters such as (/ * etc.

Good modelling practice would encourage you to create the data templates first. In practice, a lot of code writers write the model first. In this case, ICMS builds the data templates from the model symbol names.

Reiterating the naming conventions for model symbol names:

- Uppercase 1st character – a persistent model symbol which can be accessed outside the function
- Lowercase 1st character – a transient model symbol which is local to its function and cannot be accessed outside it
- Underscore 1st character – a global model symbol which can be accessed by all models in the system.

Data Types

More:

 Input Data Type

Input Data Type

Data with Data Type – Input (defined in its data template) will NOT BE UPDATED DURING EACH RUN STEP by the Main model. When the Main model is run they will have the appropriate value, but any changes to these variables will not be saved on completion of the Main model. Changes in the Initialisation and Finalisation models will be saved, however. This allows the model writer to configure input variables to be the correct size or type, and saves processing time when running the Main model.

Resolution

Numeric constants used in models are stored in the precision with which they are declared.

- Numbers with decimal places (eg. 2.5, 356.1345) are stored as doubles, which have a range of 5.0×10^{-4951} to 1.1×10^{4932} , with 15-16 significant digits.
- Numbers without decimal places (eg. 5, 201, -3527) are stored as integers, and have a range of -32768 to 32767.

This range is important to remember as numbers outside this range will be truncated during compilation, thereby making them inaccurate. For instance, if the number -32769 was used in a model, then it would be stored as 32767. To avoid this problem, the number should be specified as -32769.0 to have it stored as double precision.

Statements

A statement is a function call, assignment operation or a program control block. Function calls and assignment operations require a semicolon (;) to be added to the end of the line to signify the end of the statement. Program control blocks do not require the semicolon directly, as it is usually part of the statement(s) that follow.

More:

- [Simple Statements](#)
- [Multiple Statements](#)

Simple Statements

Examples:

```
a = b + 2;           // assignment operation
a = CurrentDay;      // assignment operation and function call
InitialiseTemporal(a, singleprecision);    // function call
```

Multiple Statements

Multiple statements can be combined together in a statement list by putting curly brackets (`{` and `}`) around the statements. This is usually done during program control blocks, and around the body of a function. If a program control block (such as an if-then-else statement) needs to execute more than one statement, curly brackets are used. If no curly brackets are used then the following statement is the only one executed.

Examples:

```
if (a == 5) b = 1; else b = 0;
```

This assigns 0 to b if a is not equal to 5, otherwise it assigns 1 to b

The semicolon is used at the end of each statement (b = 1 and b = 0).

```
if (a == 5)
{
    b = 1;
    c = 2;
}
else
{
    b = 0;
    c = 4;
}
```

This program control block needs to execute two statements per branch, so curly brackets are wrapped around the statement pairs. No semicolon is required on the end of the statement since it is a program control block.

Functions

Functions are used to break apart program structure into meaningful sections, and are commonly used when a number of statements are regularly used (ie. repeated). They can require some or no parameters, and they can return a single value. The parameters can be value, where the parameter can be modified within the function but will not be changed when the function exits, or reference, where a variable passed into a function will be updated if it is modified within the function. Reference parameters require the ampersand ('&') symbol to appear before them in the function header. Since reference parameters are updated, they must be a variable when passed. Value parameters can be either variables or constants (eg. 0, 1.5, etc).

Remember also that the MickL language prevents global variables (variables with one or more uppercase characters) from being used as parameters in functions.

There are three predefined functions: Main, Initialisation, and Finalisation. Main is run at each time step of the model run, Initialisation is run before the first time step, and Finalisation is run after the last time step. They require no parameters and return no values.

Examples:

```
function NoChange(b)
{
    b = 5;
    return 0;
}
```

```
function Change(&c)
{
    c = 4;
    return 0;
}
```

```
function Main()
{
    a = 1;
    NoChange(a);
    NoChange(1);
    Change(a);
}
```

The function NoChange has a value parameter called a. When NoChange(a) is called from the Main function, the variable b is modified to 5 within NoChange, but returns from the function as 1 since it is a value parameter. Similarly, when NoChange(1) is called from the Main function, b is assigned 1 when NoChange begins to execute, then it is assigned 5, yet no lasting changes occur. Only when Change(a) is called does a become permanently modified, since c is a reference parameter and is assigned 4. When the Change function

finishes, the value of c is assigned to a, leaving a equalling 4.

Example:

```
function Reset()  
{  
    a = 5;  
    b = 2;  
    return 0;  
    c = 1;  
}
```

The function defined is Reset, and it requires no parameters. When it is called, the variables a and b are assigned values. The curly brackets are used to identify the body of the function. The return 0 is required even though the function does not need to return a value. Since the return happens before the variable c is assigned to, this line will never be executed. Therefore return statements can be used to drop out of the current function at any point, without executing the remaining code.

Program Control Structures

More:

- if else
- for
- while
- switch
- function

if else

Test an expression for a true result, then run the following statement, otherwise run the **else** statement

if (exp) statement **else** statement

Example:

```
if (a == 1)
```

```
{
```

```
b = 2;
```

```
c = 3;
```

```
}
```

```
else
```

```
{
```

```
b = 0;
```

```
c = 0;
```

```
}
```

Example:

```
if ((d == 3) && (e != 3))
```

```
{
```

```
f = 3; // assigns f when d=3 and e<>3
```

```
}
```

*Multiple boolean statements can be used by **oring** (||) or **anding** (&&) them, and enclosing the whole in brackets*

for

Loop through the statement until finished

```
b = 1;  
for (a=0; a<5; a++)  b = b * 2;
```

The first expression is the initial conditions, the second expression is the test to keep the loop running, and the third expression is the loop counter

while

Loop through the statement until finished.

while (exp) statement

```
a = 0; b = 1;
while (a < 5)
{
    b = b * 2;
    a = a + 1;
}
```

The expression must be true for the statement to continue to run

switch

A multiple if-statement for testing a number of possible values

switch (x)

```
switch (x)
{
    case 1: b = 4;      // case 1
    case 2: b = 8;      // case 2
    case 3: b = 12;     // case 3
    default: b = 0;
```

The value in x is compared against a list of possible values, and the corresponding statement is run. Unique values are compared using the case-statement, and all others done using the default-statement

function

Defines a new function

function Name (parameters)

Example:

```
function Half(a)
{
    b = a / 2; // divides a by two
    return b; // returns the result
}

function Test(&c)
{
    c = Half(5) + Half(3);
    return 0; // no return value necessary
}

function Main()
{
    Test(c); // modifies c based on Test function
    return 0;
}
```






























The parameters can be used to pass values into the function, and can even be used to pass values out of the function by prefixing them with an & symbol.

A function must have a return value, even if it does not get used.

There are three predefined functions: Main, Initialisation, and Finalisation. Main is run at each time step of the model run, Initialisation is run before the first time step, and Finalisation is run after the last time step

MATHEMATICAL OPERATIONS

More:

-  =
-  +
-  -
-  * (scalar)
-  /
-  <
-  >
-  <=
-  >=
-  ==
-  !=
-  ||
-  &&
-  ++
-  --
-  !
-  log10
-  ln
-  exp
-  power
-  sin
-  cos
-  tan
-  arcsin
-  arccos
-  arctan
-  pi
-  rand
-  sqrt

 odd

 even

 mod

 div

 max

 min

+

Addition

$a = b + c;$

-

Subtraction

$a = b - c;$

* (scalar)

Scalar Multiplication

$a = b * c;$



Division

$$a=b/c;$$



Less than

```
if (a < b) c = 1;
```



Greater than

If (a > b) c = 2;

<=

Less than or equal to

```
if (a <=b) c = 3;
```


>=

Greater than or equal to

```
if (a >= b) c = 4;
```

==

A test without assignment

```
if ((a == 1) c = 5;
```

!=

Is not equal to

```
if ((b != 1)) c = 5;
```

||

Logical OR

```
if ((a == 1) || (b != 1)) c = 5;
```

&&

Logical AND

```
if ((a == 1) && (b == !1)) c = 10;
```

++

Increment

a++;

--

Decrement

b--;

!

Inversion

```
if (!a) c = 5; // if a is zero, then !a true, else false
```


log10

Returns the base-10 logarithm of b

```
a = log10(b);
```

ln

Returns the natural logarithm of b

```
a = ln(b);
```

exp

Returns the value of e raised to the power of b, where e is the base of the natural logarithms

```
a = exp(b);
```

power

Returns the value of base raised to the power of exponent

```
A = power(base, exponent);
```

sin

Returns the sine of x in radians

```
A = sin(x);
```

COS

Returns the cosine of x in radians

```
A = cos(x);
```

tan

Returns the tangent of x in radians

```
A = tan(x);
```

Tan(x) is the same as sin(x) / cos(x)

arcsin

Returns the inverse sine

```
A = arcsin(x);
```

x must be between -1 and 1. The return value will be in the range $[-\pi/2, \pi/2]$, in radians

arccos

Returns the inverse cosine

```
A = arccos(x);
```

x must be between -1 and 1. The return value will be in the range $[0..pi]$, in radians

arctan

Returns the inverse tangent

```
a = arctan(x);
```

pi

Returns the value of pi, which is the ratio of a circle's circumference to its diameter

```
a = sin(pi/2);
```

Pi is approximated as 3.1415926535897932385

rand

Returns a random number with the value between 0 and x

```
a = rand(x);
```

a is double precision

sqrt

Returns the square root of a number

```
a = sqrt(b);
```

a is double precision

odd

Returns true if trunc(x) is an odd number, otherwise it returns false

```
b = 5;  
if odd(b)  
{  
    c = 1;  
}    // c will be 1 since b is odd
```

even

Returns true if trunc(x) is an even number, otherwise it returns false

```
b = 6;
if even(b)
{
    c = 1;
}    // c will be 1 since b is even
```

mod

Returns the integer modulus of x divided by y

```
a = mod(5, 3);    // returns 2
```


div

Returns the integer dividend of x divided by y

```
a = div (5, 3);    // returns 1
```

max

Returns the maximum of x and y

```
a = max(5, 3);    // returns 5
```

min

Returns the minimum of x and y

```
a = min(5, 3);    // returns 3
```

MATRIX AND SCALAR FUNCTIONS

More:

- CreateMatrix
- =
- +
- -
- * (matrix)
- Transpose
- Identity
- SliceRow
- SliceCol
- MatrixHeight
- MatrixWidth
- []
- byte
- integer
- single
- double

CreateMatrix

Creates a new, or changes an existing, matrix

CreateMatrix(<matrix>,<width>,<height>,<precision>,<erase>)

```
CreateMatrix(CMat, 5, 4, BYTE, ERASE);
```

<matrix> name of resultant matrix

<width> column offset

<height> row offset

<precision> BYTE, INTEGER, SINGLE, DOUBLE

<erase> ERASE, NOERASE

If ERASE then all cells are initialised to zero, otherwise no data will be overwritten. If the array needs to expand in size then zeroes will be inserted

=

Matrix Equality

$\text{Matrix}A = \text{Matrix}B;$

Two matrices A and B are equal iff $a_{ij} = b_{ij}$ for all pairs of i and j



Matrix Addition

`MatrixC= MatrixA + MatrixB;`

The sum of 2 matrices of like dimensions is the matrix of the sum of the corresponding elements. If the matrix dimensions are not like, their addition is undefined

—

Matrix Subtraction

$\text{MatrixC} = \text{MatrixA} - \text{MatrixB};$

One matrix is subtracted from another of like dimensions by forming the matrix of the difference of the individual elements. If the matrix dimensions are not like, their difference is undefined

* (matrix)

Matrix Multiplication by a scalar

$$C = c * A;$$

e.g. A is a matrix of width 3, height 2

$$C = \begin{matrix} ca_{11} & ca_{21} & ca_{31} \\ ca_{12} & ca_{22} & ca_{32} \end{matrix}$$

A matrix is multiplied by a scalar by multiplying each element of the matrix by the scalar

Matrix Multiplication

$$AB = A * B;$$

e.g. A is a matrix of width 3, height 1 $a_{11} \ a_{21} \ a_{31}$

B is a matrix of width 2, height 3 $b_{11} \ b_{21}$

$$b_{12} \ b_{22}$$

$$b_{13} \ b_{23}$$

AB is a matrix of width 2, height 1

$$a_{11}b_{11}+a_{21}b_{12}+a_{31}b_{13} \ a_{11}b_{21}+a_{21}b_{22}+a_{31}b_{23}$$

*Also called the Dot Product. For matrix multiplication to be defined, the matrices must conform, ie the **width** of A must equal the **height** of B. AB will have dimension of width of B and height of A. BA is non-conformable, and therefore not defined*

Vector multiplication

$$AB = A * B;$$

e.g. A is a vector of width 3 $a_{11} \ a_{21} \ a_{31}$

B is a vector of height 3 b_{11}

$$b_{12}$$

$$b_{13}$$

AB is a scalar (dimension 1,1)

$$AB = a_{11}b_{11} + a_{21}b_{12} + a_{31}b_{13}$$

A 'special case' of matrix multiplication.

*Also called Vector Inner product or Dot Product and results in a **scalar***

Transpose

Transposes a matrix

```
B = Transpose(A);
```

Identity

Creates an identity matrix

Identity(<MatrixName>,<dimension>)

Identity(A,3) creates the following matrix

1 0 0

0 1 0

0 0 1

Creates a square matrix of name <MatrixName> with one in each diagonal position and zeroes elsewhere. Precision is byte, but the output from any matrix operation takes the highest precision of the input matrices.

SliceRow

Returns the elements in the specified row as a row vector

B = **SliceRow**(<MatrixName>, <row number>);

Example:

```
RequiredRow = SliceRow(InMatrix, 10);
```

SliceCol

Returns the elements in the specified column as a column vector

B = **SliceCol**(<MatrixName>, <column number>);

Use SliceCol and Transpose to graph a column

Example:

```
B = SliceCol(Results,5);
```

```
B = Transpose(B);
```

MatrixHeight

Returns the height of the specified matrix

```
height = MatrixHeight(MatrixA);
```


MatrixWidth

Returns the width of the specified matrix

```
width = MatrixWidth(MatrixA);
```

[]

Returns the matrix element at <width>,<height>

```
a = MatrixA[3, 5];
```

// returns the value of MatrixA at column 3, row 5

byte

Changes the precision of x to byte (1 byte)

```
a = byte(b) ;
```

a gets byte precision of b

integer

Changes the precision of x to integer (2 bytes)

```
a = integer(b);
```

Range -32768 to 32767. Numbers without decimal places (eg. 5, 201, -3527) are stored as integers

single

Changes the precision of x to single

```
A = single(b);
```

double

Changes the precision of x to double (4 bytes)

```
a = double(b);
```

Range 5.0×10^{-4951} to 1.1×10^{4932} , with 15-16 significant digits

GRID FUNCTIONS

ICMS has rudimentary support for grids. Grids are stored as matrices but functions operate cell-by-cell, not using matrix algebra. GRID operations have an extra parameter, NODATA (commonly -9999), which allows for a rectangular grid to represent a non-rectangular shape, such as a catchment. In the following diagram showing the multiplication operation (GridProd), the grey cells are NODATA and no processing occurs in them.

9	9	9						1	0	1						9	0	9			
9	8	9				*		0	0	1				=		0	0	9			
9	9	8	9	9				2	0	1	1	2				18	0	8	9	18	

Input grids must have the **same** dimensions. The resultant grid has the same dimensions.

Summation of a DATA cell and a NODATA cell returns a NODATA cell in the resultant grid.

If the input grids are not the same dimensions, the function returns an error

More:

- [GridSum](#)
- [GridDiff](#)
- [GridProd](#)
- [GridDiv](#)

GridSum

Grid Addition

`GridC = GridSum(GridA,GridB,NODATA)`

Sums cells, cell-by-cell, to give a new grid of the same dimensions

GridDiff

Grid Subtraction

`GridC = GridDiff(GridA,GridB,NODATA)`

Subtracts cells, cell-by-cell, to give a new grid of the same dimensions

GridProd

Grid Multiplication

`GridC = GridProd (GridA, GridB, NODATA)`

Multiplies two grids, cell-by-cell, to give a new grid of the same dimensions

GridDiv

Grid Division

GridC = **GridDiv** (GridA, GridB, NODATA)

The cells in grid1 are divided by cells in grid2, on a cell-by-cell basis, to give a grid of the same dimensions

TIME FUNCTIONS

More:

- [InitialiseTemporal](#)
- [CurrentHour](#)
- [CurrentDay](#)
- [CurrentWeek](#)
- [CurrentFortnight](#)
- [CurrentMonth](#)
- [CurrentYear](#)

InitialiseTemporal

Initialises a blank temporal with a given precision

InitialiseTemporal(<name>, <precision>,<erase>);

<name> a vector (ie matrix of height 1)

<precision> BYTE, INTEGER, SINGLE, DOUBLE

<erase> ERASE, NOERASE

ERASE - values in the resulting vector are zero-filled

NOERASE - values in the resulting vector are not cleared

Makes the variable temporal if it isn't already, then resizes it to match the width of the model run (number of time steps) by adding zeroes

CurrentHour

Returns the hour-of-day for the current time step in the current model run

```
a = CurrentHour;
```

The result is zero-based, so 1:00 am has a value of 1, 5:00 pm has a value of 17, etc

CurrentDay

Returns the Julian day-of-year for the current time step in the current model run.

```
a = CurrentDay;
```

The result is one-based, so the first day has a value of 1, and so on

CurrentWeek

Not implemented

```
a = CurrentWeek;
```

Returns zero by default

CurrentFortnight

Not implemented

```
a = CurrentFortnight;
```

Returns zero by default

CurrentMonth

Returns the month-of-year for the current time step in the current model run

```
a = CurrentMonth;
```

The result is one-based, so March has a value of 3, and so on





CurrentYear

Returns the year for the current time step in the current model run

```
a = CurrentYear;
```

TOOLS FUNCTIONS

More:

-  [ShowError](#)
-  [QuickSort](#)
-  [WhereTo](#)
-  [SimplexLP](#)

ShowError

Displays error message

Example using the condition, if a should not be negative

```
if (a < 0) ShowError(1);
```

Halts program execution if User Messages are configured to stop the program (set in Edit|Preferences|Errors/Warnings). Only 2 messages are available: 1, 'Invalid value detected'; anything else, 'General error'

QuickSort

Sorts a matrix using a quick sort algorithm (an insitu sort)

QuickSort(<matrix>,<rowcol>,<order>,<direction>)

<matrix> matrix to be sorted – is overwritten by result of sort

<rowcol> sort key row or column

<order> ASCENDING, DESCENDING

<direction> ROW, COL

ROW - sort done along the row specified by <rowcol>.

COL – sort done down the column specified by <rowcol>

WhereTo

Returns target cell when moving through matrix created by D8 flow direction tool

WhereTo (<targetx>,<targety>,<x>,<y>,<flow direction matrix>)

<targetx>, <targety> on return from the function, stores the location (width, height) of target cell

<x>,<y> location (width,height) of current cell

<flow direction matrix> name of matrix created by D8 flow direction tool

A handy function to accompany the D8 flow direction tool (which creates a 8-directional flow direction matrix and a flow order matrix from an input elevation matrix. Flow Order matrix has the same dimensions as the input elevation matrix with each cell having a value of 0-7 which indicates the target cell for the current cell. The Flow Direction matrix stores the sequence (width, height of cells) to cycle through the elevation matrix.)

WhereTo is a set of switch/case statements which returns the target cell reference, depending on whether the cell order is 0 or 7.

SimplexLP

Solves a Linear Model by optimising using the simplex linear programming algorithm.

SimplexLP(<objectivefunction>,<constraints>,<maxmin>,<finalcost>,<coeffs>)

<objectivefunction> a $[n,1]$ dimensional matrix of n coefficients representing the coefficients of the terms on the RHS of the equation

<constraints> an $[(n+2),m]$ dimensional matrix where n is the number of coefficients in the LHS of the constraint equations and m is the number of constraints.

The first n columns contain the left hand side coefficients, the $n+1$ column contains numbers representing the <equality statements>, and the $n+2$ column contains the right hand side coefficients.

Equality statement values are:

< -2

<= -1

= 0

>= 1

> 2

<maxmin> MAXIMIZE - maximise the objective function

MINIMIZE – minimise the objective function

<finalcost> on return from the function, stores the minimised or maximised value of the objective function, ie the final cost

<coeffs> on return from the function, stores the coefficients used in the objective function that match the <finalcosts>

The return value of the function will be 0 (ie false) if no solution could be found, otherwise it will be 1 (ie true).

If the Simplex algorithm cannot minimise or maximise the objective function (usually due to an inappropriately defined problem), the program will log an Other error, which may halt the program if Edit|Preferences|Errors/Warnings has been appropriately configured.

Form of objective function:

Z =

$$\sum_{j=1}^n c_j x_j$$

Form of constraints:

$$\begin{aligned} \sum_{j=1}^n a_{1,j} x_j &\leq (\geq) b_1 \\ \sum_{j=1}^n a_{2,j} x_j &\leq (\geq) b_2 \\ &\vdots \\ \sum_{j=1}^n a_{m,j} x_j &\leq (\geq) b_m \end{aligned}$$

Example: Solve $Z=20x_1+15x_2$, subject to

$$x_1 \leq 100$$

$$x_2 \leq 100$$

$$50x_1+35x_2 \leq 6000$$

$$20x_1+15x_2 \leq 2000$$

<objectivefunction> is a $[2,1]$ matrix containing $[20, 15]$

<constraints> is a $[4,4]$ matrix containing

$$1 \ 0 \ -1 \ 100$$

$$0 \ 1 \ -1 \ 100$$

$$50 \ 35 \ -1 \ 6000$$

$$20 \ 15 \ 1 \ 2000$$

Example:

```
if (SimplexLP(objfn, consts, MAXIMISE, final, coeffs))
{
    // use the result
}
else
{
    // something went wrong
}
```

Tools

More:

- What are They?
- Why are They Needed?
- Accessing Tools
- Writing Tools
- Available Tools

What are They?

Tools are stand-alone programs which provide additional functionality to ICMSBuilder. They are not available within (and are totally separate from) models and are commonly associated with data processing. They are invoked manually by selection from the Tools menu.

Tools and Views have much in common. They can only be written by a skilled programmer in the programming environment used to develop ICMSBuilder, namely Inprise® Delphi. They are compiled as dynamic link libraries (DLLs) and stored in the \Plug-Ins subdirectory. A protocol for Tool writing is available which defines the procedures that must be encoded in the Tool to ensure its correct linkage with ICMSBuilder.

Why are They Needed?

Tools are useful because they greatly expand the range of functionality that ICMSBuilder provides, without a need to modify and recompile the core ICMS product, ICMSBuilder. They give ICMS programmers a means to distribute their particular visualisation or data transformation algorithm within ICMSBuilder.

Of course, many Tools may start their life as MickL code within a model. However, this compromises the notion that an ICMS model only contains 'model code' (with ICMS handling all other overheads). If it (ie a set of MickL code) is only performed at the start or the finish of a run, and the user is happy to access the Tool manually from the menu, then it may be implemented as a Tool. If it is required iteratively within a model run, then it may be implemented as a MickL function

This can only be done within ICMSBuilder. Send an email to icms@cbr.clw.csiro.au with suggestions.. Both these solutions require a skilled Delphi programmer.

Accessing Tools

Tools are accessed via the Tools menu. ICMSBuilder searches the \Plug-Ins subdirectory and builds a list of the Tools (.DLLs) that it finds there.

Writing Tools

Tool writers are supported by written documentation and templates for different styles of Tools. These are available from the ICMS website. Two documents are provided:

- Tasks for Designing a New DLL Tool in ICMS (ICMS Working Document #68)
- Delphi Compiler Settings for ICMS (ICMS Working Document #65)

Once a Tool is written and compiled, it is immediately available to ICMSBuilder, simply by storing it in the \Plug-Ins subdirectory. For a Tool to be made available to the ICMS community, a Grant of Licence must be completed (available from the ICMS website). Then the Tool will be loaded to the website and made available for download. It may also be included in any future ICMS distribution disks.

Available Tools

ICMSBuilder v2 has 2 tools distributed with it:

- Raster D8 Flow algorithm
- Flow statistics

More:

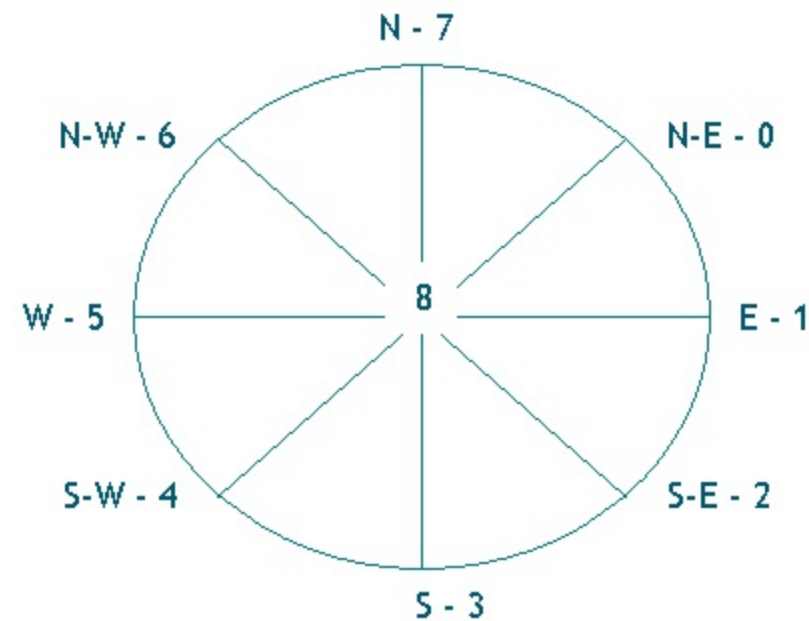
■ [Raster D8 Flow Algorithm](#)

■ [Flow Statistics](#)

Raster D8 Flow Algorithm

This tool reads an elevation matrix and creates two matrices to assist with its processing – namely a matrix of flow direction for each cell, and a matrix of flow routing order. The data templates for the output matrices must exist before the tool is invoked. The output matrices are persistent, ie the output is stored in the named data instances.

The flow direction matrix takes the same dimensions as the elevation matrix, and each cell contains integer values from 0 to 8, representing



flow directions from the current cell, with 8 being no flow (ie. the flow does not leave the cell since all neighbours are higher). These values are the direction to the lowest neighbouring cell.

The flow routing matrix has the dimensions of [width*height, 2], ie its width is equal to the cell count of the elevation matrix, and its height is two). This matrix is filled with the (x, y) pairs of cells that represent the highest to the lowest cells in the input elevation matrix.

It is complemented with a WhereTo function in MickL which returns the cell location (width, height) of the receiving cell for each cell.

Example:

Data instance Elevation, 6x8 matrix

	1	2	3	4	5	6
1	167	165	152	132	134	129
2	144	151	156	146	132	126
3	150	141	139	135	131	126
4	158	151	153	146	136	128
5	158	160	160	153	148	141
6	155	157	157	156	152	149
7	152	153	154	150	144	140
8	147	150	153	151	149	147

Select Tools|D8 Flow Algorithm. A dialog box will appear for you to enter the input raster name, and the names of the two output matrices. While the output matrices can have any name, we use FlowDirection and FlowOrder.

Raster D8 Flow Algorithm

Data

Input DEM
(m by n matrix)

Catchment (1): Elevation: 6x8 - matrix, double

Output Flow Direction
(m by n matrix)

Catchment (1): FlowDirection: 6x8 - matrix, byte

Output Flow Order
(mxn by 2 matrix)

Catchment (1): FlowOrder: 48x2 - matrix, integer

Start

Close

?

Help

Press the Start button. A Done message box will appear when the processing is complete.

Contents of the output FlowDirection and FlowOrder matrices are

	1	2	3	4	5	6
1	3	4	1	8	2	3
2	2	3	3	1	1	8
3	1	1	1	1	1	8
4	0	7	7	7	1	7
5	0	7	0	0	0	7
6	3	3	2	2	2	3
7	3	4	1	1	1	8
8	8	5	5	0	0	7

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
1	1	2	3	3	2	2	1	3	3	1	2	1	1	2	1	3	4	3	2	1	3	2	1	2	4	4	5	6	5	4	3	5	4	6	5	4	4	4	5	5	5	6	5	6	6	6	6	6
2	1	1	2	1	2	5	4	5	4	5	4	3	2	6	6	6	6	7	7	7	8	8	8	3	2	5	6	6	5	4	3	1	1	5	4	3	8	7	2	3	8	8	7	1	2	4	3	7

Here is a sample of code which uses the tool and the WhereTo function in a temporally dynamic model. As well as the D8 flow algorithm matrices (FlowDirection and FlowOrder), it uses another matrix, CellWater, to store model results.

```
function Main()
{
  // get the cells to process
  height = MatrixHeight(CellWater);           // same as input elevation matrix
  width = MatrixWidth(CellWater);
  cells = width * height;
  // loop through the cells from top
  for (count = 1; count <= cells; count++)
  {
    currentx = FlowOrder[count,1];    // offset (col, row) of current cell
    currenty = FlowOrder[count,2];

    // code here to do whatever processing is required within the current CellWater cell

    WhereTo(targetx,targety,currentx,currenty,FlowDirection);
    If (FlowDirection[currentx,currenty] != 8)           // internally draining
    {
      CellWater[targetx,targety] = CellWater[targetx,targety] + outflow;
    }
  }
  return 0;
}
```

Flow Statistics

This tool reads a daily time series (a **temporal** matrix of increment **Days**) and produces a series of useful statistics and graphs. The output is not persistent, ie they only exist for the duration that the tool is open and are not stored within the named objects.


The tool is based on a set of hydrologic-descriptors considered useful for ecological assessment and are derived from Young (1998)

Young, WJ. (1998) Hydrologic Descriptions of Semi-Arid Rivers: An Ecological Perspective. In: Kingsford, RT (ed). "Free-flowing river: the ecology of the Paroo River". Proc. of the Paroo Scientific Workshop, 1997. NSW National Parks & Wildlife Service.

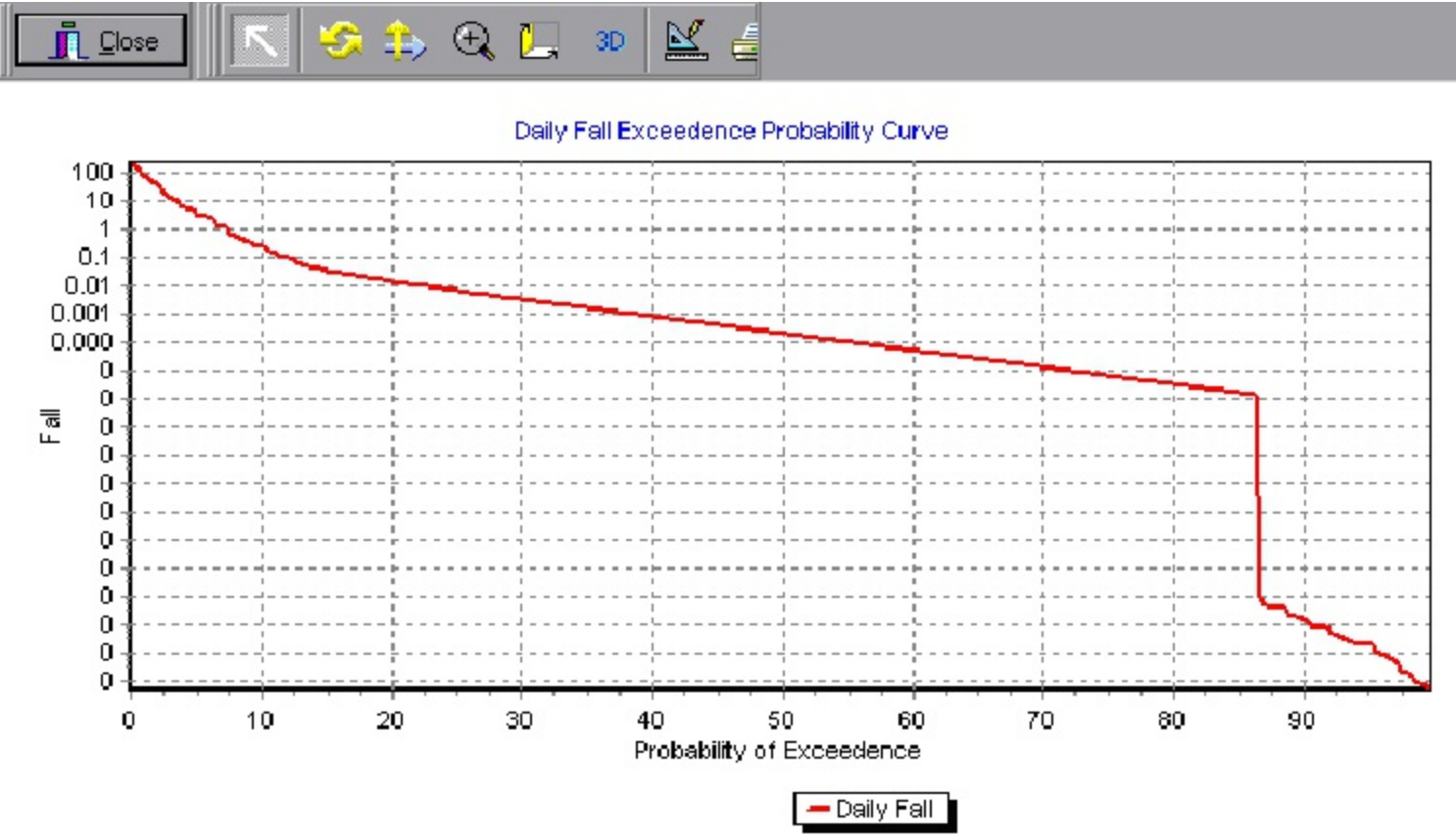
The tool has only been implemented for daily data.

Select a Daily Time Series	Monthly Data
419027ih - Outflow	Median Value
	S80
	Minimum
	Maximum
	Time Series Data...
	Exceedence Probability...

Daily Data	Annual Data
Median Value	Median Value
S80	S80
Minimum	Minimum
Maximum	Maximum
Time Series Data...	Time Series Data...
Exceedence Probability...	Exceedence Probability...
Rise Exceedence Probability...	
Fall Exceedence Probability...	
Peak Values Probability...	
Low Spells Probability...	

The tool builds up a list of daily temporal matrices within the current project. These can be cycled through by selecting the  button in the 'Select a Daily

Time Series' window. Here is an example of the Daily Fall Exceedance probability curve. You need to press the Close button at the top of the chart to return to the Tool dialog.



Views

More:

- [What are They?](#)
- [Why are They Needed?](#)
- [Views and Projects](#)
- [Accessing Views](#)
- [Designing Project Views](#)
- [Writing Views](#)

What are They?

Views are stand-alone programs which provide a tailored interface to an ICMSBuilder project. A View is opened in ICMSBuilder, after you have opened its project. A project can have many views – and a view can be associated with many projects, provided those projects have the same structure, just different data.

A View is a tool for interrogating a project, using a different interface to that provided by ICMSBuilder. It has access to the functionality within ICMSBuilder and adds its own functionality.

Views are not written by an ICMS user in MickL code. They are written by a skilled programmer in the programming environment used to develop ICMSBuilder, namely Borland® Delphi. Views are compiled as dynamic link libraries (DLLs) and stored in the \Plug-Ins subdirectory. A protocol for View writing is available which defines the procedures that must be encoded in the View to ensure its correct linkage with ICMSBuilder.

Why are They Needed?

ICMSBuilder provides the modeller's view of a project. It provides access to every object, every variable, and every model. It provides general interfaces to do this – the system view, the numeric view for viewing and editing scalars and matrices, the raster view for viewing and editing rasters and grids, the graph view for viewing and editing matrices, and the class and object information views.

There are times when a tailored view is required, particularly when the project is being delivered to a third party, such as a catchment management group. It is one way of rapidly building interfaces which are specific to a particular set of management and/or scientific issues.

Views and Projects

ICMS Views are always tightly associated to their underlying project. This is because they usually refer to specific data in specific objects. An exception would be a View which builds a project.

A water allocation project and a View for that project are available from the ICMS website (`WaterAllocationExample.icm` and `WaterAllocationView.dll`). This View is very tightly bound to the objects of the underlying data.

Accessing Views

Views are accessed via the View|Project Views menu. ICMSBuilder searches the \Plug-Ins subdirectory and builds a list of the Views (.DLLs) that it finds there. While this list contains all the Views that you have placed in that subdirectory, only some (or none) may be relevant to the project that you have open. The View writing protocol gives guidance on coding the View such that it will not function (or is at least benign) if an inappropriate project is open.

Designing Project Views

An advantage of this approach to interface development is that the underlying system representation – classes, models and objects - are not driven by the style of the interface. The underlying structure can be worked over many times before any decision is made about what components of the project should be made available to an audience, other than the model developer. In fact, they can be altered after the View is designed, provided the data and object names referenced by the View do not change.

Thus, the project developer can take the whole system to the audience, work through all or part of it. They can discuss with the client for the View their needs, what parts of the system they would like to 'play with', and how they would like to see results presented. During this phase, the tools within ICMSBuilder provide sufficient functionality for them to understand and view the whole workings of the project.

Another advantage of this approach is that many Views can be rapidly built for the one project. This means that the same project can be presented in many different ways, appropriate to the intent of the View (educational, training, trade-off negotiations, technology transfer, etc).

Writing Views

View writers are supported by written documentation and templates for different styles of views. These are available from the ICMS website. Two documents are provided:

- Tasks for Designing a New DLL View in ICMS (ICMS Working Document #45)
- Delphi Compiler Settings for ICMS (ICMS Working Document #65).

Once a View is written and compiled, it is immediately available to ICMSBuilder, simply by storing it in the \Plug-Ins subdirectory. For a View to be made available to the ICMS community, a Grant of Licence must be completed (available from the ICMS website). Then the View will be loaded to the website and made available for download.

VIEWS, MENUS, TOOLBARS, and DIALOG BOXES

More:

 [ICMS Views](#)

ICMS Views

ICMS has a number of views that are used to navigate through and manipulate the data. These views are central to ICMS and implement the class/object paradigm. The views are:

- [Class Information](#) view
- [Object Information](#) view

In true Windows tradition, there are many ways of working with these views. For example, to edit an object, you can use the System View, the Object Information View or the Object Data View. These views are described below. After that, menus and toolbars which work with these views are described.

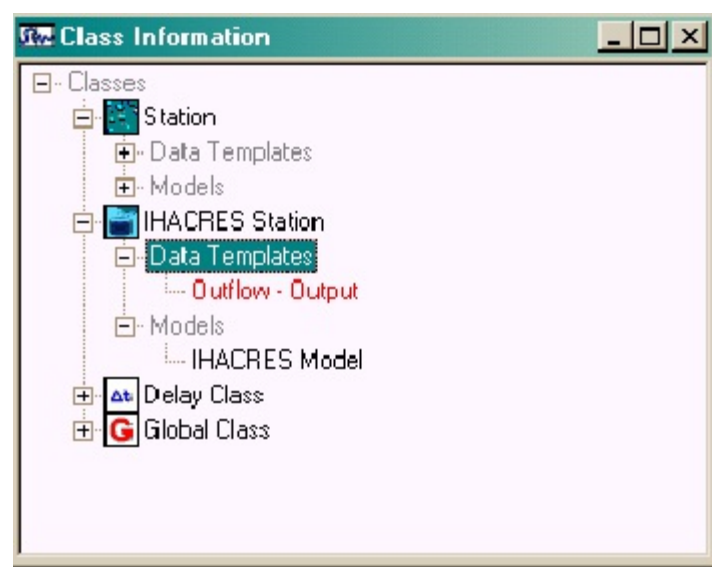
More:

■ [Class Information](#)

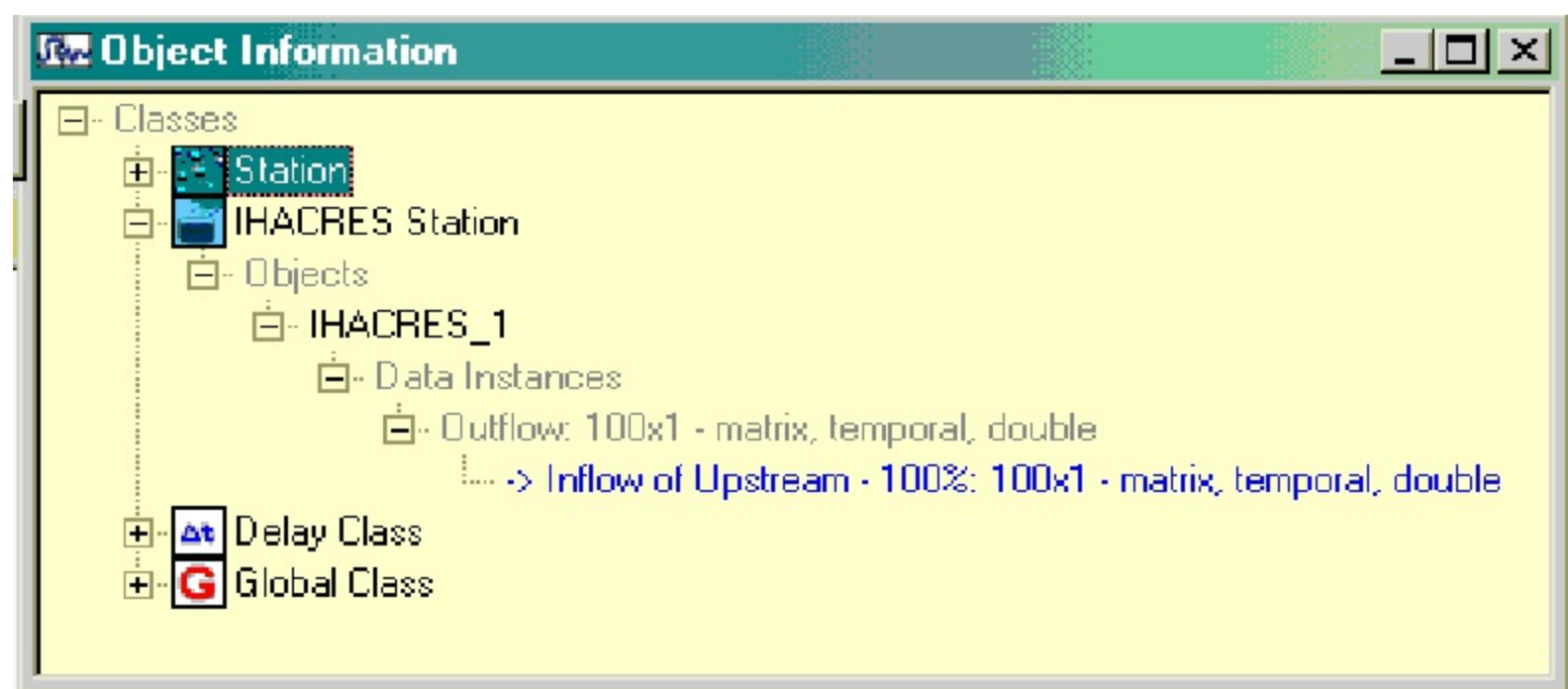
■ [Object Information](#)

Class Information

The class information view is described in the Working with Classes section.



Object Information



The Object Information Tree is very similar in its structure to the Class Information Tree. Each class is listed under the Class node. Under each class is listed all its objects (the objects that were **instantiated** from the class). (The above figure shows the objects associated with the class IHACRES Station). For more information about objects, read **WORKING WITH OBJECTS**.

toolbar. Data instances that link to or from data instances in other objects are shown with **->** or **<-** symbols and the name of that data instance, along with the percentage of the link. The **->** symbol represents a link to the destination, and the **<-** symbol represents a link from the source.

Parse results are created when the model is run and you have chosen to save some results. For instance (using the data in the diagram above), if you wish to compare the value of **Outflow** for one run against another, you would select **Outflow** as a Save Result and after each run **Outflow** would be saved as a parse result. Each parse result can be viewed in the same way as a data instance, in either a numeric, map or graph view.

Component objects define how each object relates to other objects. Every object belongs to another object. In the case of the top level objects, they belong to the **ICMS Root Object**, as special object which stores global data and the root level of all objects. Any of these top level objects can **own** other objects, thereby creating a **hierarchy** of objects. This is particularly useful when compartmentalising your system. Say your system has multiple sub-systems operating at different spatial locations (eg. A complex water balance model running in various discrete parts of the catchment). At each location you can create a top level object representing the sub-system. By **zooming**

into this object you will get a blank system view to work in. In this view you can create multiple objects, link them together, and feed their results back to the top level (parent) object. This saves cluttering the top level view and allows you to focus on each sub-system without worrying about the others.

As this is a tree structure you can open and close any branch of the tree by clicking on the **+** or **–** boxes.

The right mouse button is used to access many functions in this view by right-clicking on a particular node in the tree. Only those menu options which are valid for the particular node will be enabled. These functions are also available through the Data menu.

This view is opened using View|Object Information Tree.

Troubleshooting

The following section details some common problems and their solutions.

More:

- [Output doesn't change on Run](#)
- [Nothing happens on run](#)
- [Problems with passing data in and out of Parent Objects](#)
- [Selecting Classes when Importing a Model Library](#)
- [`You must choose a valid file first!' in Data Import Wizard](#)

Output doesn't change on Run

Reason. Output may be defined as an INPUT. In this case, even though it is calculated, ICMS doesn't update the value.

Solution. Change to OUTPUT (or INPUT/OUTPUT or LOCAL). This is a property of the data template and is updated using the Data Template Properties dialog box – accessible from the Classes/speed menu of the Class Information View

Nothing happens on run

It may be that you have used locals (model symbol names commencing with lower case) to define important variables. If you have done this in **Initialisation**, these values are not available in **Main**. The model will compile, but you won't be accessing the values.

In the following example, width has no value in **Main**, and looping does not occur.

```
function Initialisation()  
{  
    width = MatrixWidth(LandUseRaster);  
    return 0;  
}  
  
function Main()  
{  
    // loop through the land use matrix and store in cell load matrix  
    for (i=1; i<=width; i++)  
    {  
        // heaps of interesting code here!  
    } // end of i loop  
    return 0;  
}
```

Remedy: Put **width = MatrixWidth(Matrix);** at the start of **Main**.

Problems with passing data in and out of Parent Objects

By definition, parent objects contain no models. Most of these data will be received from another object and passed down to a child; or received from a child and then passed onto another object. Thus all data templates for a Parent Object need to be defined as **Input/Output**.

Selecting Classes when Importing a Model Library

While you can select individual classes (by ticking their box), all classes are imported. This will be remedied in v3 (post 2001).

‘You must choose a valid file first!’ in Data Import Wizard

Of course, this message may indicate exactly what it says – you have selected a file of the wrong file format. However, if you are sure that you have the right file format, then it is probably because you have the file open. If this is the case, simply close the file, and try again. The import routine doesn’t discriminate on the error type when it attempts to read the input file.

ICMS Tutorials

5 tutorials are provided to lead you through the features of ICMS. Tutorial 0 shows you how to build a project from an existing set of classes and model libraries. Tutorials 1, 2, 3, and 4 build up from a static single catchment model to a spatially and temporally distributed subcatchment model. Tutorial 5 is just for fun and is an implementation of a cellular automata game - Conway's game of life. It is included to show the versatility of the ICMS environment for building different types of models.

Tutorial 0 builds up an ICMS project and shows you how to:

- view classes
- create objects
- add data and models to objects
- link objects
- run the models
- view results

This is all without writing a single line of code!

Tutorial 1 builds a simple (single) catchment model and shows you how to:

- create a class (as a template from which to build objects)
- create a model that can be linked to your class (and hence objects) so that data can be processed
- create the object(s) from the class
- add data to the individual object(s)
- run the model
- view the results.

Tutorial 2 builds on Tutorial 1 by linking together subcatchment objects and passing data out of one object into another (downstream). This tutorial leads you through how to:

- create more classes and a new model
- use global data instances
- create and link multiple objects
- add data to the objects
- include a second model
- create and insert routing objects
- run the combined system
- view the results.

Tutorial 3 introduces time and builds a simple flow routing model (using Muskingum routing). It shows you how to:

- load the muskingum routing equation (a model template)
- run the model and view the results
- use delta objects (used as a form of time delay between linked objects)
- build a regression model.

Tutorial 4 covers spatially distributed processing using a DEM. It introduces you to:

- a class/object with a 3D data template
- a model for moving water over a 3D surface
- building a temporal model
- using the Time Manager to run the model
- viewing temporal data.

Tutorial 5 is not really a tutorial at all! It simply challenges you to use ICMS for a very different sort of problem – a cellular automata problem. No worked solutions are provided!

More:

- Source of Tutorials materials
- Tutorial 1 - Creating a simple nutrient load model
- Tutorial 2A - Creating a dIstributed nutrient load model
- Tutorial 2B - Add Instream Assimilation Model
- Tutorial 3 - Temporal data, deltas and routing models
- Tutorial 4 - Runoff routing on an elevation raster
- Tutorial 5 - Cellular automata problem
- Exercise 1. Building a project
- Exercise 2. Simple nutrient load Model
- Exercise 3. Routing material through a catchment

Source of Tutorials materials

Data, projects, and model libraries used in the tutorials are distributed with ICMS. The actual name of the ICMS directory and its location depend on you installation. The installation procedure defaults to ICMS vXXX (where XXX is the version number, eg 2.1) located in the Program Files\ directory.

The tutorial material resides in:

ICMS vXXX

Tutorials

Projects (.ICM files)

ModelLibraries (.MDL files)

Data (Input data files)

Tutorial 1 - Creating a simple nutrient load model

In this tutorial you will create a simple model of total nutrient load exported from one catchment containing a range of land uses. This demonstrates how a model is created and run in ICMS.

The model is based on the Catchment Management Support System (CMSS) model. This model requires two input parameters - the areas of the different land uses and nutrient generation rates for those land uses. Total nutrient load is then calculated by multiplying areas by generation rates. The area of each land use is stored in an array. (Land use areas may be 0). Generation rates for nutrients from those same land uses are stored in another array. The model finds a single nutrient load for the catchment by calculating the dot product of these two arrays (which multiplies cells and adds them up in the one process).

As you go through the tutorial, pay attention to the sequence of steps required to create and run a model. You will follow these steps every time you create a modelling structure in ICMS.

The steps are:

[STEP 1 - Create a project](#)

[STEP 2 - Creating a Class](#)

[STEP 3 - Add Data Templates](#)

[STEP 4 - Creating a Model](#)

[STEP 5 - Creating objects](#)

[STEP 6 - Add Data](#)

[STEP 7 - Run](#)

Good Modelling Practice

We encourage you to follow good modelling practice by using sensible and meaningful data template and model symbol names, and by creating your data templates before you write your models. ICMS will create data templates for model symbol names which have not been previously referenced. However, this is sloppy design and is not encouraged.

More:

- [Materials used in this Tutorial](#)
- [Create your Working Directory](#)
- [STEP 1 - Create a project](#)
- [STEP 2 - Creating a Class](#)
- [STEP 3 - Add Data Templates](#)

- Viewing the Class
- STEP 4 - Creating a Model
- STEP 5 - Creating objects
- STEP 6 - Add Data
- STEP 7 - Run

Materials used in this Tutorial

- Data file Tutorial1.csv
- Project file Tutorial1Done.icm
- Project file Tutorial1 (+ext).icm
- Recommended reading
- Working with Classes
- Working with Objects
- Working with MickL.

Create your Working Directory

You can work in the ICMS Tutorials directories. To ensure that you don't overwrite any ICMS projects, you may prefer to create your own working directory. The following instructions assume that you have created a directory called `c:\ICMSWorking`.

STEP 1 - Create a project

To create a model, it is first necessary to create a project. This is done by selecting File|New. For this exercise name it Tutorial1 in \ICMSWorking directory. Press OK. A new project will be created.

*Note: A finished version of this tutorial is in
.\Tutorials\Projects\TUTORIAL1DONE.ICM.*

STEP 2 - Creating a Class

There are many ways of creating a class. Here we use the Add Class wizard which is available from the Classes menu or from the Class Information speed menu (select Classes and press the right hand mouse button).

On Page 1, create a class called Subcatchment by typing Subcatchment into the Class Name text box. It is possible, although not required, to add a description of the class.

On Page 2, associate an icon with the class. You can use the default or browse through the ICMS vXXX\Icons sub-directory.

STEP 3 - Add Data Templates

On Page 3, add the following three data templates by following the wizard's instructions:

LandUseLoads (of type Output)

GenerationRates (of type Local)

- LandUseAreas (of type Local).

Note: Take care with the format of names - the MickL language in ICMS is case sensitive. Names of objects or classes within the ICMS program should contain only alphabetic or numeric characters, with the stipulation that the first character cannot be a number.

More:

■ [Adding Models](#)

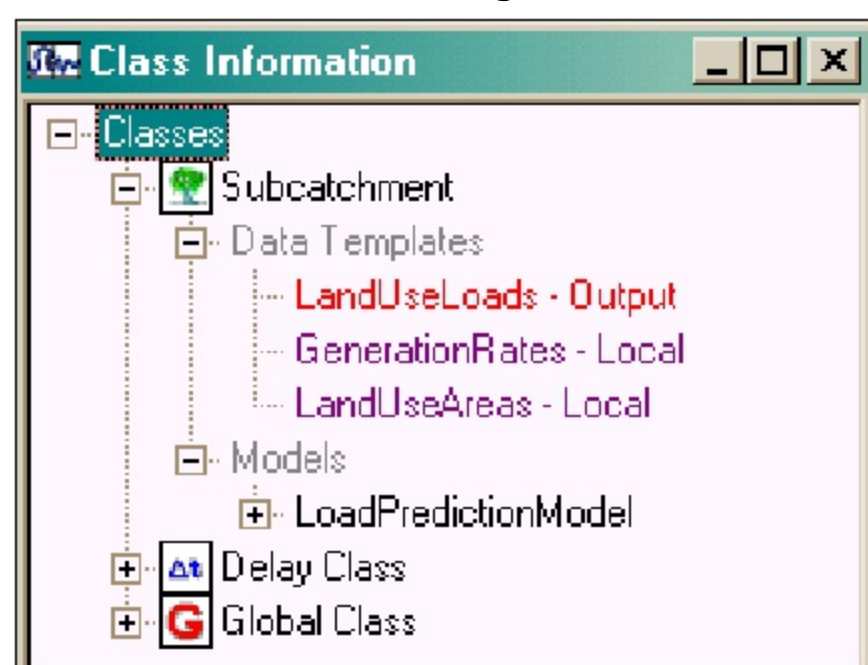
Adding Models

At this stage, you can only add the name of the model that you will be coding. You can do this now or when you are ready to code your model. If you do it now, on Page 4 enter LoadPredictionModel. On Page 5, press Finish to complete the creation of the class.

Viewing the Class

To see this new class, open the Class Information view. This view will show all the classes.

The two classes (Delay Class and Global Class) are managed by ICMS and are described in the Working with Classes reference sheet.



STEP 4 - Creating a Model

So far you have created a class that represents an area of land or a Subcatchment. This class is like an object factory that builds Subcatchment objects. Each of these objects will have their own data instances, ie values that use the GenerationRates, LandUseAreas and LandUseLoads data templates. To sum the products of different generation rates and land uses we need to write a MODEL.

More:

- [Adding Model Formula](#)

- [Compiling a Model](#)

Adding Model Formula

In ICMS a model belongs to a class. Here LoadPredictionModel belongs to the Subcatchment class.

Select Edit Model Formula (double click on the model name in the Class Information view). The model editing window will open for you to type MickL code.

Enter the following lines of code: (// represents a comment line).

Hint: You can pick from a list of model symbol names and functions by clicking on the right hand mouse button. For further information about MickL, go to Help|Contents.

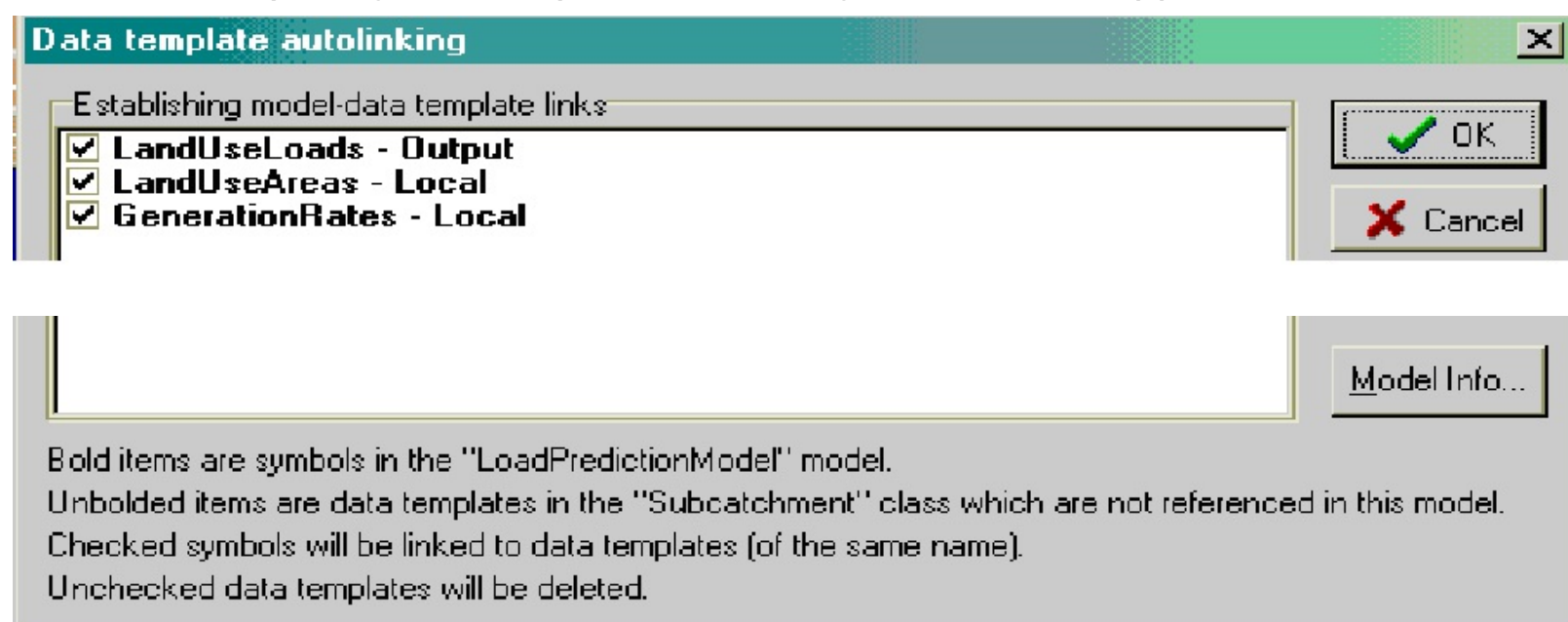
Enter **LandUseLoads** = 0; in Initialisation()

Enter **LandUseLoads** = **GenerationRates** * **LandUseAreas**; in Main()

What are we doing here? The model code presupposes that the data is stored in matrices. The model multiplies the generation rates and land use area matrices to produce another matrix.

Compiling a Model

Before ICMS can run this model, it needs to be compiled to check for syntax errors. The quick way to start the compiler is to press the **F9** key. If you entered everything correctly, the following window will appear:



While compiling the model, ICMS will automatically link model symbols to data templates. Press **OK** to automatically connect the model symbols with the data templates in the class. This links the model to your class, allowing you to pass data to your model when you run it.

To close the Formula Edit dialog box, press the cross in the top right corner.

If the Data template autolinking window doesn't appear, you can look in the messages below the model view to see what error was found. You can press Cancel and return to the model editing window (Edit model formula) to correct any syntax errors. When the errors have been corrected press **F9** again to recompile the model.

Of course, running the model now won't do anything because you haven't defined any objects which use this model; nor have you loaded any data for the model to operate on.

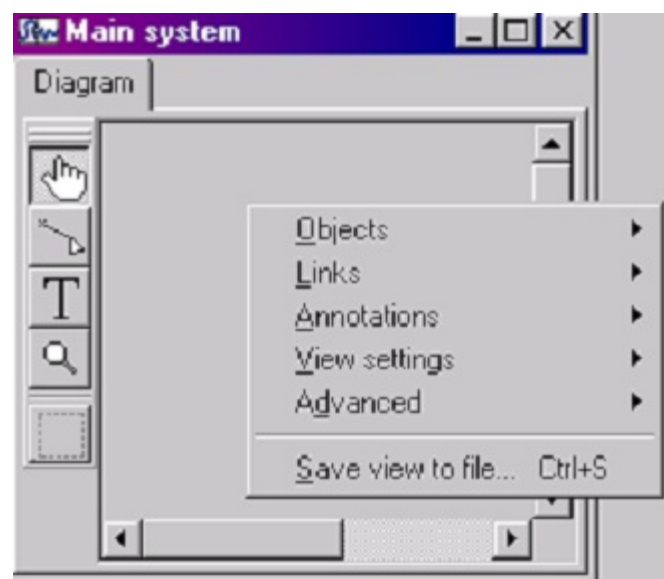
STEP 5 - Creating objects

Now that you have a class with a model linked to it we can proceed to create an object from the Subcatchment class. The System View provides a visual way of doing this. (You can also use the Object Information view).

Open the System View via the icon on the toolbar.

Right click anywhere on the pane and the System speed menu will appear.

Select Objects|Add. The Object|Properties window will appear.



You want to create a Subcatchment object called **Head Creek** and you want it to use your Model. To do this, overtype New Object with Head Creek, select Subcatchment from the Class drop-down box, and

select LoadPredictionModel from the Model drop-down box. Press **OK** to create the object.

For another view, let's look at the Object Information View (.)

STEP 6 – Add Data

Now we can add data to Head Creek. You have a table that looks like this:

Land Uses	Generation Rates (kg/ha/yr)	Area (ha)
Bushland	0.1	5
Established Sewered Urban	1.3	15
Unsewered Peri-Urban	0.6	22
Industrial & Commercial	1.8	87
Fertilised Grazing	1.25	50
Unfertilised Grazing	0.25	5
Disturbed Land	20.0	110
Piggery	0.3	14
Established Unsewered Urban	5.3	9
Build Up- Miscellaneous	1.8	43

You have already created places to hold these data, namely GenerationRates and LandUseAreas. The Head Creek object inherited these data instances automatically from their class. For this exercise, we need to define these data instances as matrices to exploit the power of matrix operations. There are many ways of adding data. We will show 2 ways – using the Data Import wizard; and using the Numeric View.

More:

- [Using the Data Import Wizard](#)
- [Using the Numeric View](#)

Using the Data Import Wizard

The Data file is provided as a comma delimited file, Tutorial1Data.csv, in the ICMS vXXX\Tutorials\Data sub-directory.

File | Import | Data import wizard

On Page 1, use Browse to find the file and select Comma Separated Variable (CSV) format.

We can only import one data item at a time. To import the generation rates, on Page 2 follow the choices shown in this figure. ICMS does not handle strings – so you must skip the 1st column and row. Because ICMS graphing works on rows, let’s transpose the data.

Now you can select what sort of file this is.

Format

☒ Comma delimited

☐ Tab delimited

☐ Semicolon delimited

☐ Space delimited

Start import at row

Start import at column

☒ Transpose data

☒ Only one row

	1	2	3	4	5	
1	0.1	1.3	0.6	1.8	1.25	0.2

On Page 3, leave at double precision.

On Page 4, select GenerationRates in Head Creek to import the data into. On Page 5, press Finish.

Format

- ☒ Comma delimited
- ☐ Tab delimited
- ☐ Semicolon delimited
- ☐ Space delimited

Start import at row

3

Start import at column

2

☒ Transpose data

☐ Only one row

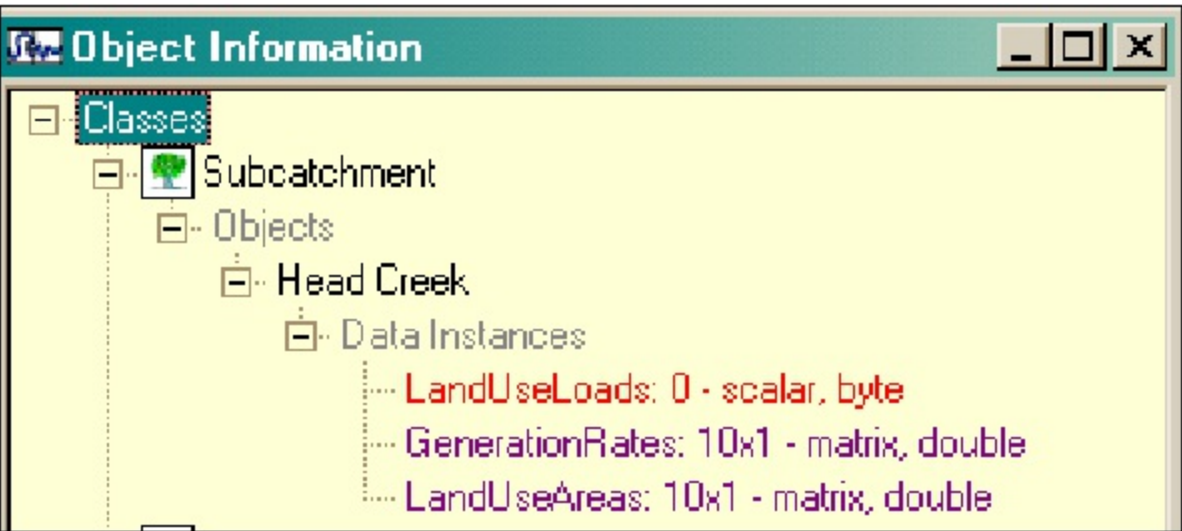
	1	2	3	4	5	
1	5	15	22	87	50	5

Now repeat the exercise to import the LandUseAreas. On Page 2, follow the choices shown in this figure.

On page 4, select LandUseAreas in Head Creek.

Using the Numeric View

Even if you have already imported the data using the Data Import Wizard, we can use the Numeric View to look at the data and explore the editing facilities. There are so many ways of getting to a view of the data! Let's use the Object Information View .

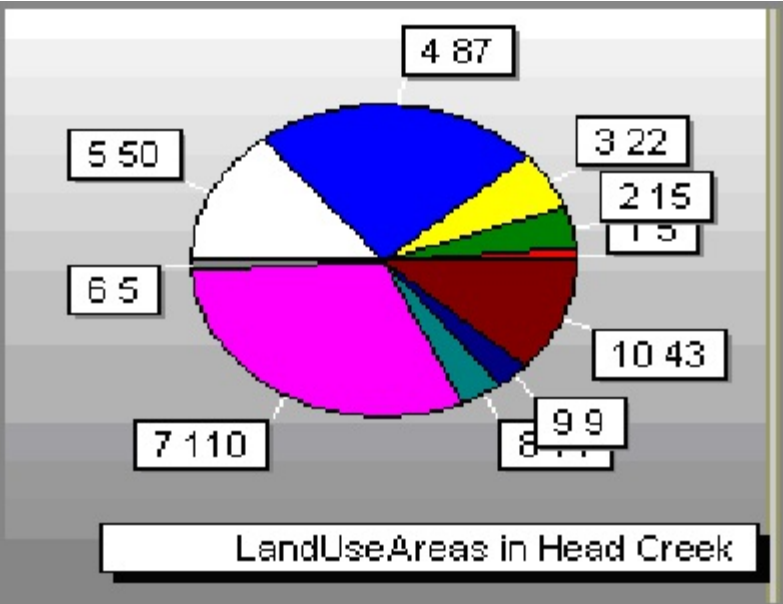
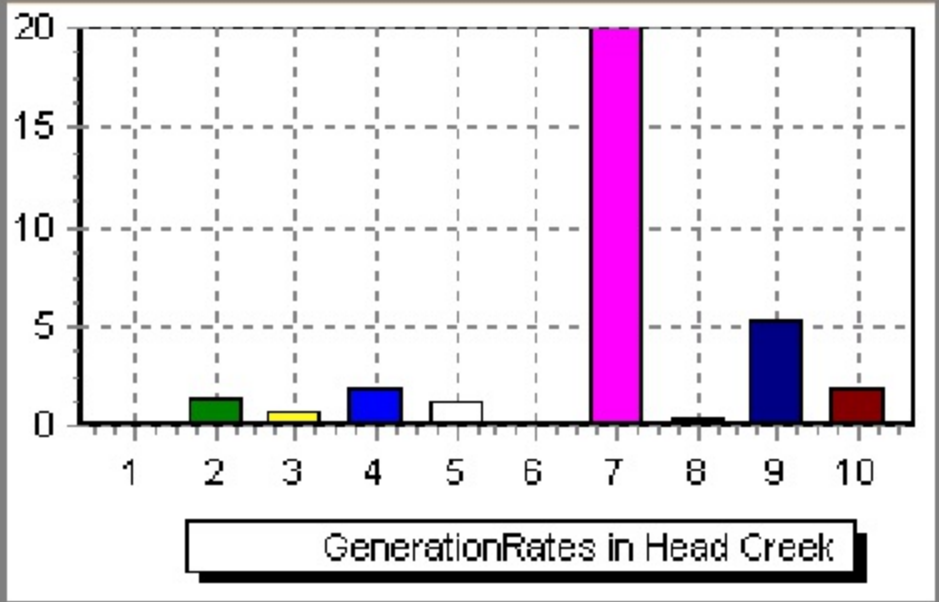


Double click on GenerationRates. Because you defined that the default view was numeric when you created the data template, the Numeric View will appear.

Notice that LandUseLoads is currently defined as a scalar with byte resolution. This will change once you do a run.

GenerationRates in Head Creek - 10 by 1 matrix										
	Scalar	Matrix	Resize	Temporal	3 Byte	1032 Integer	1.5 Single	1.1213 Double		
	1	2	3	4	5	6	7	8	9	10
1	0.1	1.3	0.6	1.8	1.25	0.25	20	0.3	5.3	1.8

For alternate views (raster or graph), right click on GenerationRates. Select Values | Graph.



More:
■ [Sizing a Matrix](#)

Sizing a Matrix

The Data Import Wizard sized the matrix from the data. Otherwise, use the Resize button on the Numeric View. Do that now.

The Numeric View works like a spreadsheet. You edit the values in the cells.

STEP 7 - Run

Having supplied all the data the Model is now ready to be run. Before running the model, open a view of LandUseLoads (double click on LandUseLoads in OIV will show as numeric; or right click | Value | View value as a). It will be empty at this stage, as it represents the output or result of the model we are about to run.

To start a run, click on the Run Manager icon . As there is no temporal data in our project, you will only see a Start button. (Later we will see what happens for a temporal run.)

Press Start and view LandUseLoads. If you entered the same numbers as above it should look like:

Congratulations, you have produced your first working model in ICMS.

More:

■ [Extension 1 to Tutorial 1](#)

Extension 1 to Tutorial 1

What if we now want to total these separate land use loads to have a total load exiting the subcatchment?

The simplest way is to transpose one of the matrices and use the dot product (refer to the Functions section of Working with MickL reference).

In this extension exercise, you will also see how ICMS responds when you add a new name in your code before you have added its data template; learn about local variables; use an ICMS function.

Open the LoadPredictionModel formulae editor (OK, it's up to you now – use either the Class Information View or right click on the object)

Initialisation(); add

TotalSubcatchmentLoad = 0;

Main(); add

tempmatrix = Transpose(**GenerationRates**);

TotalSubcatchmentLoad = **LandUseAreas** * tempmatrix;

Compile (F9).

Look at the Class Information View. You will see a new data template for TotalSubcatchmentLoad. ICMS has created this as the template for the new variable you have created because it has an uppercase 1st character.

Change its type to Output (required if you ever want to transfer this data to another object).

You will notice there is no data template for tempmatrix. Why? It has a lowercase 1st character. ICMS knows this as a local variable, ie it only exists during the model run.

If you want to look at this matrix, simply change its 'spelling' to Tempmatrix. This will create a data template of type Local which can be viewed.

OK, open a view of TotalSubcatchmentLoad. This single number is the dot product of the transpose of GenerationRates and LandUseAreas, meaning that our subcatchment produces 2582.85 kg/yr of nutrient.

Imagine what you can do when you start linking subcatchments together and accumulating LandUseLoads and TotalSubcatchmentLoad through the system?

Tutorial 2A - Creating a dIstributed nutrient load model

Introduces you to cloning models, global variables, linking objects, using model libraries to add new models, and using the Run Library to compare scenarios

In this tutorial we expand on the simple nutrient load model developed in Tutorial 1 to make it capable of being distributed across a large catchment.

In Tutorial 1 we estimated loads for a subcatchment in isolation. To capture the effect of these nutrients being exported from an upland catchment to a lower catchment we would like to run our model upstream, then pass the result down to another subcatchment, before it calculates its own load and subsequently passes it on.

To achieve this we need to include:

an input to the model for upstream subcatchments to be routed through, and an assimilation model to model instream reduction of the nutrient load.

In other words we need to adjust the model to say that the load for a subcatchment is a combination of what is generated in the subcatchment, plus what is passed down to it from upper catchments.

You will also learn how to prevent duplicating data by using global data templates.

Good Modelling Practice

We encourage you to follow good modelling practice by using sensible and meaningful data template and model symbol names, and by creating your data templates before you write your models. ICMS will create data templates for model symbol names which have not been previously referenced. However, this is sloppy design and is not encouraged.

More:

- [Materials used in this Tutorial](#)
- [Steps - Tutorial 2](#)
- [Steps - Tutorial 2b](#)
- [Working Directory](#)
- [STEP 1 - Cloning a Model](#)
- [STEP 2 - Create and edit global data](#)
- [STEP 3 - Edit the Model](#)
- [STEP 4 - Copy objects](#)
- [STEP 5 - Link objects](#)

■ STEP 6 - Import data

■ STEP 7 - Run

■ STEP 8 - View results

Materials used in this Tutorial

- Data file ICMS vxxx\Tutorials\Data\Tutorial1Data.csv
- Project file ICMS vxxx\Tutorials\Projects\Tutorial1 (ext1).icm
- Project file ICMS vxxx\Tutorials\Projects\Tutorial2Done.icm
- Project file ICMS vxxx\Tutorials\Projects\Tutorial2bDone.icm
- Model library file ICMS
vxxx\Tutorials\ModelLibraries\AssimilationModel.mdl
- Recommended reading
- Working with Classes
- Working with Objects
- Linking Objects
- Working with MickL.

Steps – Tutorial 2

- Clone a model
- Create and edit global data
- Edit the model
- Copy subcatchment objects
- Link subcatchment objects
- Import data
- Run
- View results

Steps – Tutorial 2b

- Import a class (and its model) from a model library
- Create an instream object and link it between subcatchments
- Add data
- Run and view results

Working Directory

Stay in your working directory. Open your project file from Tutorial 1, or, if you didn't do the Tutorial 1 extension exercise, open the project file ICMS vXXX\Tutorial1 (ext1).icm. Save as Tutorial2 in your working directory.

STEP 1 - Cloning a Model

We begin by updating the LoadPredictionModel. To prevent losing our previous version of this model, we will use the Clone feature. Right click on LoadPredictionModel in Class Information and choose Clone model. This will copy the model formula and open an editor, allowing you to start modifying the model code.

STEP 2 - Create and edit global data

Remember that the `GenerationRates` variable holds the nutrient generation rates for a set of land uses. In this example, these are the same for all subcatchments. Is there a way of declaring them only once? Yes, rather than store `GenerationRates` in each object, let's make it a **global**. These are special data instances and are coloured pink in the model editor. Making a variable global is simple – simply place an underscore `_` in the 1st character of `GenerationRates` (ie `_GenerationRates`) in the model code. You will notice its colour change to reflect its new global status. (You will need to compile the model at this stage to create the data template and data instance.) We cannot encourage good modelling practice here! This is the only case where you cannot add a data template directly to the Class. The global class is managed by ICMS.

In the data template autolinking window, that comes up after compiling, you will notice that `GenerationRates` is no longer bold. This indicates that it is no longer present in the model code. Even though the variable is no longer used, it is not automatically deleted as sometimes you may need a variable in one model but not in another. Leaving it around saves you from redefining it when you change models.

The next step is to resize the new global `_GenerationRates` to `[10,1]` and give it some data.

Tip: To resize a matrix, use the Numeric View. Globals are 'owned' by the ICMS Globals object

Alternatively (and much easier!), just copy Generation Rates to `_Generation Rates` by using Copy and Paste.

This will copy the contents and resize in the same operation.

Add data to the resized matrix if you haven't done so. Let's view `_GenerationRates`.

You can now safely delete `GenerationRates`. If you decide to delete it, you can simply uncheck its box in the autolink dialog next time you compile, or explicitly delete it from the Class Information View.

STEP 3 – Edit the Model

Add InSubcatchmentLoad and InLandUseLoads to pass nutrient data from one object to another. We start to get into slightly more complicated stuff here. Firstly, we need to be more careful with defining the resolution of the data instances. The matrix multiplication of GenerationRates * transpose(LandUseAreas) results in a 1,1 matrix. While this is to all intents and purposes a scalar, it cannot be added to a scalar. So, we need to make a few changes from the 1st version of the model.

```
function Initialisation(). Define a few data instances
width = MatrixWidth(LandUseAreas);
CreateMatrix(LandUseLoads,width,1,DOUBLEPRECISION,ERASE);
CreateMatrix(TotalSubcatchmentLoad,1,1,DOUBLEPRECISION,ERASE);
CreateMatrix(InSubcatchmentLoad,1,1,DOUBLEPRECISION,ERASE);
CreateMatrix(InLandUseLoads,width,1,DOUBLEPRECISION,,E
ERASE);
```


function Main() change the formulae to include the upstream load in the calculation.

```
LandUseLoads = InLandUseLoads + LandUseAreas * _GenerationRates;
tempmatrix = Transpose(_GenerationRates);
TotalSubcatchmentLoad = InSubcatchmentLoad + LandUseAreas *
tempmatrix;
```

Notice in the code that we have defined the new data instances in *Initialisation()* rather than in *Main()*. Instructions in *Initialisation()* are executed before *Main()* begins. We specifically use it to set aside space in memory for variables like InLandUseLoads.

Compile this new model to check for errors.

Tip: F9 compiles a model when the editor window for the model is in focus.

Let's rename the model LoadPrediction Model v2 (right click on Model name from CIV; Edit model). Open a System View by pressing  on the toolbar or by View|System. You should already have one object in the view called Head Creek. (*This view allows you to place multiple objects in two dimensions and link them.*) The Subcatchment object is still linked to the old model from Tutorial 1. To link it to the new model, double click on the object, and choose the LoadPrediction Model v2 in the model drop-down-box.

STEP 4 - Copy objects

So far you have cloned the model, changed the model, and have redefined **_GenerationRates** as a global, giving it some data. Remember the goal here is to have objects talking to each other. Before we go any further, we need to create the objects that will do the talking.

Next we will create three more subcatchments, Subcatchment2, Subcatchment3 and Subcatchment4 (or whatever you want to call your subcatchments!).

We will use the Clone Object function. Open a System View. Select Head Creek, right click, select Objects | Clone Object. And so on. Move the objects to locations that match the configuration of your imaginary landscape.

STEP 5 - Link objects

Now the objects have been created, we need to link them together.


To make things easier for us we can define the InSubcatchmentLoad and InLandUseLoads as of type Input. (Of course, if we had followed good modelling practice, we would have defined their data templates first!). Go to the CIV – double click on InSubcatchmentLoad. The data template properties box will appear. Because the template was created by the Compiler, the default data type is Local. Change this to Input. While you have the CIV open, do the same to InLandUseLoads.

Your CIV should now look like the following figure.

(Notice the use of different colours to differentiate between output, local and input data templates. This is mirrored in the Object Information View.

You can change the colours to suit you via the Edit | Preferences | Colours..)

Why have we bothered to change the data type to Input and Output? Because it makes establishing Links that much easier. Only Input and Output data templates can be linked – so they are the only ones that appear in the Link dialog box.

Press  on the left toolbar of the System View. This enables the linking mode. Now click and drag a link from Head Creek to SubCatchment2.

We want LandUseLoads from Head Creek to feed into InLandUseLoads in Subcatchment2. We also want TotalSubcatchmentLoad from Head Creek to feed into InSubcatchmentLoad in Subcatchment2.

Select LandUseLoads and drag across, with the mouse still depressed, to

InLandUseLoads. Release the mouse. You will see the link appear in the Links box. Do the same thing for the other link. You have now established 2 links.

You can use the same Links dialog box to establish all the other links. Simply select a new set of From and To objects. Go on. Link all the subcatchments together.

Once you have finished linking, remember to return the System View to select mode (click on).

Look and see how the links are displayed in the Object Information View.

STEP 6 - Import data

We now have a system of linked Subcatchment objects, and global generation rates to drive the models. However, all the Subcatchments have the same land use data. This is OK for this tutorial. However, if you want to use other data, you can import columns of data from the Tutorial1Data.csv file.

There are many ways of getting to the Import Data menu. One way is to open the Object Information view, right click on LandUseAreas in Subcatchment2, and select Value | Import wizard. (Details on using the Import wizard are in Tutorial 1).

STEP 7 - Run

Run! It may be more interesting if you open views of some of the Output data instances and see how they change during a run. Of course, the run is so fast because it is doing a very simple calculation without having to worry about managing temporal data.

STEP 8 - View results

Why not look at the output from each subcatchment in a multiple input graph (on the tool bar).

Tutorial 2B – Add Instream Assimilation Model

More:

- [STEP 9 - Import Model Library](#)
- [Step 10 - Create and Link an Assimilation Object](#)
- [Step 11 - Add Data to Assimilation object](#)
- [Step 12 - Run and View Results](#)
- [Extension 1 to Tutorial 2b - A Quick Intro to the Run Library](#)

STEP 9 – Import Model Library

Now we will do some instream modelling to "assimilate" the nutrient, depending on the length and shape of the stream. The traditional way to add another model is to append the new model to the original one, thereby making a "super" model. A more flexible approach is used in ICMS where different models can be connected together in the system view.

The new model can be imported by selecting File|Import|Model Library... and choosing the `Tutorials\ModelLibraries\AssimilationModel.mdl` file. This brings in the model and data templates used in the Routing class.

The instream assimilation model attached to the InstreamAssimilation class is a simple exponential law with two parameters.

- K is a measure of cross-sectional stream shape or depth, and
- T is a measure of the travel time for the water between subcatchments.

By including this model object between each Subcatchment we can simulate the loss of nutrient from the water flow while travelling along a length of river.

Once you press OK, the InstreamAssimilation Class will appear in the Class Information View.

Step 10 – Create and Link an Assimilation Object

Let's insert an object of this class between Subcatchment3 and Subcatchment4. Call it Routing 3t4. Make it of the InstreamAssimilation class using the AssimilationModel.

Now link TotalSubcatchmentLoad of subcatchment object to Input of assimilation object. Link Output of assimilation object to InSubcatchmentLoad of the downstream subcatchment object.

Step 11 – Add Data to Assimilation object

Before we can run this model, we need to provide some data. Try 5 and 0.3 for K and T respectively.

Step 12 – Run and View Results

Run.

What, the model doesn't run? Why? Look at the OIV for Subcatchment4. ICMS can't add a scalar to a matrix. We need to modify the Initialisation code in the Assimilation Model from

Output = 0; to CreateMatrix(Output,1,1,DOUBLEPRECISION,ERASE);

Note also in the OIV that we have 2 loads being accumulated in Subcatchment4. Of course, we should have removed the link between subcatchment3 and subcatchment4. However, this shows how easy it is to add or remove links without having to make any changes to the model code. Remove the link.

Check that the output of Subcatchment4 has been reduced by the appropriate amount due to the operation of the assimilation model. This is best done by comparing the input of the assimilation object to its output.

Extension 1 to Tutorial 2b – A Quick Intro to the Run Library

ICMS has the facility to retrieve and compare results of model runs using the Run Library. Each run stores input, state and output data in the object's data instances. In addition, you can select particular data instances to store in Saved Data in the Run Library. Let's explore it using Tutorial 2b.

Go to the Run menu and select Data to Save. Scroll down to Subcatchment4. Tick TotalSubcatchmentLoad. We also want to store the critical input variables that have affected this result. scroll down further to the assimilation object and tick K and T.

Go to the list in Selected Items to Save. Select each entry in turn and change Save When to 'Save after Finish'.

Press OK. Another dialog box will appear which gives you the opportunity to name the run.

Run. You can retrieve the results from View | Run Library. Check out the Object Information View to see how the Saved Data is accessible from there.

Tutorial 3 – Temporal data, deltas and routing models

Introduces you to the Temporal Run Manager, time delay objects, and temporal data

Tutorial 3 demonstrates how to model a simple hydrological flow routing in ICMS. It introduces **delta** objects which are a form of time delay between linked objects in a system.

In Tutorial 2 we linked together some Subcatchment objects and passed data out of one object into another. If the model from Tutorial 2 is extended to run over several time steps, the output from each object would be passed directly to the receiving model, on the next time step. There are many situations where this does not represent the physical system we wish to model. For instance water travelling down a stream may take longer than one time step to reach the physical location we wish to model it at. In these situations the **delta** object is useful to act as a delay in the system.

Good Modelling Practice

We encourage you to follow good modelling practice by using sensible and meaningful data template and model symbol names, and by creating your data templates before you write your models. ICMS will create data templates for model symbol names which have not been previously referenced. However, this is sloppy design and is not encouraged.

More:

- [Materials used in this Tutorial](#)
- [Steps - Tutorial 3](#)
- [Working Directory](#)
- [STEP 1 - Open the Project](#)
- [STEP 2 - The Muskingum Routing Model](#)
- [STEP 3 - Running the Model](#)
- [STEP 4 - Viewing the Results](#)
- [STEP 5 - Exchanging the Muskingum Model for a Linear Regression Model](#)
- [STEP 6 - Adding Delta \(Time delay\)](#)
- [STEP 7 - Run with different Input values and Different Time Periods](#)

Materials used in this Tutorial

- Project file ICMS vxxx\Tutorials\Projects\Tutorial3.icm
- Data file Tutorials\Data\Inflow.val.
- Data file Tutorials\Data\UpstreamTrueFlow.val
- Data file Tutorials\Data\DownstreamTrueFlow.val.
- Recommended reading
- Working with Classes
- Working with Objects
- Linking Objects
- Working with MickL.

Steps – Tutorial 3

- Open the project
- Investigate the Muskingum routing model
- Run the model
- View the results
- Exchange models (replace Muskingum model with regression model)
- Add a time delay object
- Run the model for different time periods (playing with the Temporal Run Manager)

Working Directory

Open the project file ICMS vXXX\Tutorial3.icm. Save as Tutorial3 in your working directory.

(The project file contains the input flow data (which is exported from the IHACRES object), observed flow data for upstream and downstream stations, and Muskingum constants for Upstream and Downstream stations.

However, if you want to familiarise yourself (again) with importing data, data can be imported as follows:

flow data - from Tutorials\Data\inflow.val.

Upstream:TrueFlow – from Tutorials\Data\Tut3 UpstreamTrueFlow.val

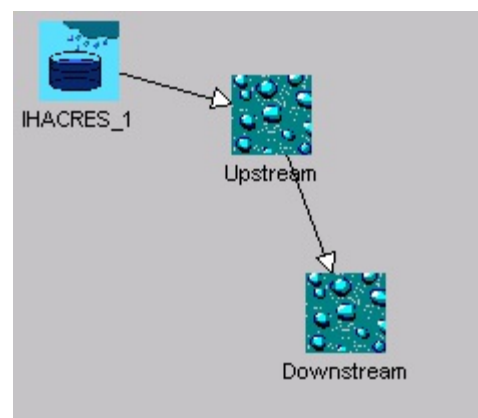
Downstream:TrueFlow - from Tutorials\Data\Tut3 DownstreamTrueFlow.val.

The Muskingum values are listed later in this document.)

STEP 1 – Open the Project

Open up the following 3 windows:

- Class Information (View | Class Information)
- Object Information (View | Object Information)
- Main system (View | System View)



The System View shows an uppermost station **IHACRES_1** that outputs simulated flow data to the object Upstream. Upstream and Downstream are Station objects and route flow according to the model Routing Model.

Familiarise yourself with the classes, data templates, objects, data instances and models.

STEP 2 – The Muskingum Routing Model

The Routing model is an implementation of Muskingum Routing equations.
The Muskingum Routing equation is:

Outflow(t) = C0 * Inflow(t) + C1 * Inflow(t-1) + C2 * Outflow(t-1)

where:

Outflow(t) is Outflow at time t

Inflow(t) is Inflow at time t

Inflow(t-1) is Inflow at time t-1

Outflow(t-1) is Outflow at time t-1

C0, C1, C2 are Coefficients determined according to:

$C2 = \exp(-t/K*(1-X))$ (Coeff²)

$C1 = K*(1-C2)/t - C2$ (Coeff¹)

$C0 = 1-(K*(1-C2)/t)$ (Coeff⁰)

where:

	Description	Upstream object	Downstream object	
K	is a storage constant	1.5	1.8	
X	is a weighting factor ranging between zero and unity	0.7	0.7	
t	is time (in days)	1	1	

K and X are determined outside of ICMS and remain constant for a given reach.

STEP 3 – Running the Model

Let's run the model and view its output. To run the model open the Run Manager (Run | Start),. The model will begin running automatically.

Tip: To speed up the model run, close the Object and Class Information Views!

STEP 4 – Viewing the Results

To view the results of the model run

- open the Object Information
- double click on Upstream - Outflow (this shows a graph because we have set graph as the default view in its data template)

Tip. If you are wanting to graph multiple data instances on the one graph, you can use the multiple graph icon and select the data instances.

Another way of getting multiple values on a graph, is simply to select another value, say Upstream: Trueflow and drag it onto the Graph window to compare the True and Simulated Flows for that station.

STEP 5 – Exchanging the Muskingum Model for a Linear Regression Model

One alternative to the Muskingum Routing model is a Linear Regression Model. In this step, we select a Regression Model to replace the Muskingum Model. The regression model requires us to create some delay objects and link them to our system.

Using Regression Modelling the relationship between Inflow and Outflow for a station can be simply represented as:

$$\text{Outflow} = \text{MyCoef} * \text{Inflow}$$


where: **MyCoef** is a coefficient determined using Ordinary Least Squares Regression.

Let's replace the Muskingum routing model with the regression model for both Station objects. How?

- Double click on the objects in the System View. select the Reg model (instead of the Routing Model).
- In the Object Information Tree right-click on the Downstream object and select Edit Object.
- In the Model text field select Regression Model instead of Routing Model and click OK.
- Do this for the Upstream object also.

STEP 6 – Adding Delta (Time delay)

So far we have not seen or used a **delta** or time delay object as Muskingum Routing includes the time delay in the model variable **t**. However more often than not it will be useful to simulate the delay of one or more time steps rather than coding it into the model. For example, one alternative to the Muskingum Routing model is a Linear Regression Model. The regression model requires us to create some delay objects and link them to our system.

To add a delta object, use the System View. Add two delta object  and link them between IHACRES and Upstream, and Upstream and Downstream. Don't forget to delete the direct link between Upstream and Downstream.

This link dialog shows you how to link Upstream to delay2.

Your System View should now look like this:

STEP 7 – Run with different Input values and Different Time Periods

As previously, to run the model open the Run Manager (Run | Start), and press the Start button.

Try changing the value for MyCoef and re-run the model. To do this select MyCoef for one of the stations and double click.

To edit more than one scalar, use Edit scalar Values for on the objects menu. You access them by either right click on the object in the system View, or on the object in the Object Information view.

This is a rather nice general tool which gives you access to all scalars for an object. It is probably not useful to edit an output data instance!

You can chose to run a shorter time-span if you wish. To do this, you need to have more control over the Run Manager. So, instead of starting the run by Run | Start, use the Run icon from the toolbar.

This displays the Temporal Run Manager window.

Play with changing the Start and Finish Dates. You do this by

- (1) the 'video' controls
- (2) the drag controls

Congratulations, you have finished Tutorial 3. You are now an advanced ICMS user!

```

* Model:      Routing Model
* Class:      Station
* Cat File:   RoutDemo.cat
* Date:      6-Sep-1999
* Author:    N. J. Ardlie / CSIRO Land & Water
*****/

```

```

function Initialisation()
{
/**
 * Determine the muskingum coefficients for the routing.
 * @param t is the time delay (in units of days)
 * @param K is the storage delay time
 * @param X is the weight for the weighted average discharge ( from the set{x:0<=x<=1})
 **/
t = 1;
Coef2 = exp(-t/(K*(1 - X)));
Coef1 = (K*(1 - Coef2)/t) - Coef2;
Coef0 = 1 - (K*(1 - Coef2)/t);
InitialiseTemporal(Outflow, DOUBLEPRECISION, ERASE);
InitialiseTemporal(RoutingError, DOUBLEPRECISION, ERASE);
Counter = 0; // counter is used in Main to determine step 1 re: Outflow calculation
Last_outflow = 0;
Last_inflow = 0;
return 0;
}

```

```

function Main()
{
  if(Counter == 0) //ie: on day 1...
  {
    Outflow = 0;
    Last_outflow = Outflow;
    Last_inflow = Inflow;
    Counter++;
    return 0;
  } // end if
  else
  {
    Outflow = Coef0*Inflow + Coef1*Last_inflow + Coef2*Last_outflow; // Muskingum Routing
    RoutingError = Outflow - Trueflow; // stores the routing error for comparative purposes
    Last_outflow = Outflow; // stores todays Outflow for use in tommorows calculations
    Last_inflow = Inflow; // stores todays Inflow for use in tommorows calculations
    if(Outflow < 0 ) Outflow = 0; //flow shouldn't be negative but Last_outflow can stay 0
  } // end else
  return 0;
}

```

```

/*****
* Model:      Routing Model
* Class:      Station
* Cat File:   RoutDemo.cat
* Date:      6-Sep-1999
* Author:    N. J. Ardlie / CSIRO Land & Water
*****/

```

```

function Initialisation()
{
/**
 * Determine the muskingum coefficients for the routing.

```

```

* @param t is the time delay (in units of days)
* @param K is the storage delay time
* @param X is the weight for the weighted average discharge ( from the set{x:0<=x<=1})
**/

```

```

InitialiseTemporal(TrackInflow,SINGLEPRECISION,ERASE); // matrix view of inflow
InitialiseTemporal(Outflow, SINGLEPRECISION, ERASE);
InitialiseTemporal(RoutingError, DOUBLEPRECISION, ERASE);
return 0;

```

```

}

```

```

function Main()

```

```

{

```

```

    Outflow = Mycoef * Inflow; // the regression relationship
    RoutingError = Outflow - Trueflow; // stores the routing error for comparative purposes
    TrackInflow = Inflow;
    return 0;

```

```

}

```

Tutorial 4 – Runoff routing on an elevation raster

Introduces you to ICMS support for processing gridded spatial data

This tutorial demonstrates how a model can transport data spatially to represent processes in the real world. Rainfall will be applied to a 3D surface, and runoff generated which can either soak into the ground or travel to the lowest surrounding cell, depending on the system constants.

At the end of this tutorial, is a short description of an application that has been built on the tools described in this tutorial.

Good Modelling Practice

We encourage you to follow good modelling practice by using sensible and meaningful data template and model symbol names, and by creating your data templates before you write your models. ICMS will create data templates for model symbol names which have not been previously referenced. However, this is sloppy design and is not encouraged.

More:

- [Materials used in this Tutorial](#)
- [Steps - Tutorial 3](#)
- [Working Directory](#)
- [STEP 1 - Open the Project](#)
- [STEP 2 - Process the Input Elevation Raster](#)
- [STEP 3 - Run](#)
- [Blackwood Application](#)

Materials used in this Tutorial

- Project file ICMS vxxx\Tutorials\Projects\Tutorial4.icm
- Data file Tutorials\Data\Inflow.val.
- Data file Tutorials\Data\UpstreamTrueFlow.val
- Data file Tutorials\Data\DownstreamTrueFlow.val.
- Recommended reading
- ICMS Tools
- Working with MickL.

Steps – Tutorial 3

- Open the project
- Investigate the data format and model
- Use the D8 Flow Routing tool – an elevation raster processing tool
- View the results

Working Directory

Open the project file ICMS vXXX\Tutorial4.icm. Save as Tutorial4 in your working directory. This catchment contains a single object, Catchment, of the Runoff class. Look at the model in this class to see how it moves water around the soil water matrix.

STEP 1 – Open the Project

Check out the System View and the Object Information View. In this project, the System View is rather uninteresting as all the processing is done inside the model code.

The Runoff class has 6 data templates which are inherited by the Catchment object. These are:

Rainfall	time series of daily rainfall
ElevationRaster	Input elevation raster (20 x 20 matrix)
FlowDirection	Matrix of flow direction (0-8) for each cell (20 x 20 matrix). Output of the D8 Flow Routing tool. (Refer to OnlineReference – Tools for details).
FlowOrder	Matrix of x,y of sequential cell processing. Output of the D8 Flow routing tool
SoilWater	Output of model (20 x 20 matrix)
Constant1	Constant which governs the transfer of water across the landscape. Tutorial is distributed with 0.2
Constant2	Constant which governs the transfer of water off the surface into the groundwater. Tutorial is distributed with 0.2.

STEP 2 – Process the Input Elevation Raster

If you are using the Tutorial4.icm that was provided with ICMS, you don't really have to import these data as that has already been done. However, it is useful to learn to use the tools that are provided to work with grids of elevation.

Data for the elevation raster is stored in Tutorials\Data\dem.txt. It is a 20 by 20 raster of elevation (m). This was imported into the ElevationRaster data instance of the Catchment object using the Import Wizard. You can overwrite the current contents if you want to do a refresher on the Import wizard.

The elevation raster is then processed using the D8 Flow routing tool. Refer to OnlineReference – Tools for details.

Use the data templates, FlowDirection and FlowOrder, as outputs of the tool.

STEP 3 – Run

To run the model, open the Temporal Run Manager . You can run it for either all or part of the rainfall time series. When you have chosen the run time, press the Start button.

Since this is a temporal model it is a good idea to open a few data views to see what is happening (though this does degrade performance!)

The important one is SoilWater since this is the output, but it is also good to view the input – eg Rainfall and ElevationRaster to determine what should be happening.

Play with the functionality of the Raster View. Its is in prototype stage only. You can change the colour scheme by right clicking on the Colour Scheme box.

The top icons are primitive scenario generators – ie you can change cell values. However, this is not fully implemented.

Blackwood Application

Author: Dr John Gallant, CSIRO Land and Water, 29 September 1998

More:

- [Structure of the original Blackwood model](#)
- [Model components](#)
- [Implementation details](#)

Structure of the original Blackwood model

The Blackwood AEAM model is designed to allow catchment stakeholders to explore scenarios for managing surface water quality, groundwater rise and surface salinisation. The model simulates the surface water balance, groundwater movement and surface salinisation on a monthly time step with a spatial resolution of 1km. It is written in Visual Basic and contains about 6 pages of code divided into 5 subroutines that manage the simulation process and implement various parts of the model.

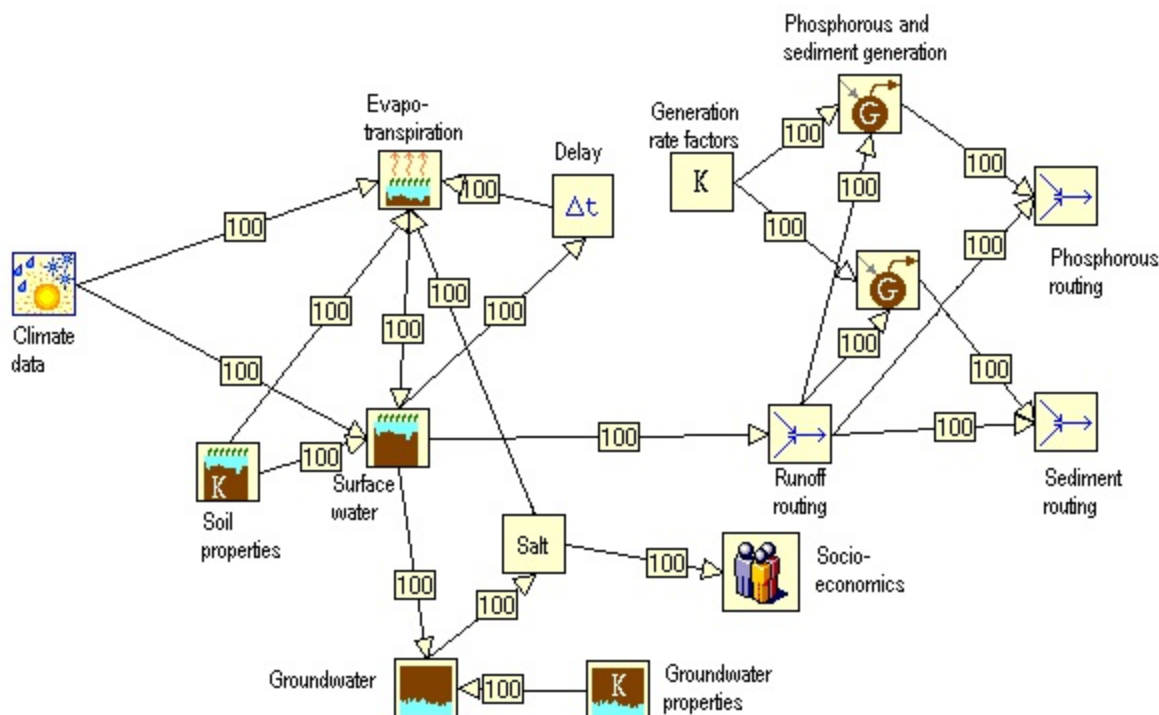
The model data comprises a large matrix of data describing the properties of each 1km² cell (including elevation, land use, vegetation cover, initial salt level, surface flow direction) plus a set of tables describing such properties as the evaporation factors for various crop types and the extent and types of cropping in different regions.

Model components

Implementation of the ICMSBuilder version of the Blackwood model commenced by identifying the components of the system model:

- surface water balance
- evapotranspiration
- groundwater recharge, diffusion and pumping
- salinisation
- surface runoff routing
- sediment and phosphorous generation by surface runoff
- sediment and phosphorous routing

Each of these components requires inputs from, and delivers outputs to, other components of the system at each time step. These links between components are created using a visual model building tool within ICMSBuilder. The view below is a screen snapshot of the linked components (with descriptive text added).



Note that the routing of runoff, sediment and phosphorous is accomplished by three separate instances of a single type (or class) of component, a routing component. This component was written once and used three times for routing the three fluxes (water, sediment and phosphorous).

Implementation details

Each of the components identified was implemented individually, tested using synthetic data then incorporated into the combined system model. The modelling environment provides simple mechanisms for exporting complete components and importing them into another system which facilitates this mode of development.

ICMSBuilder provides tools for viewing spatial and temporal data and for managing the duration of dynamic simulations based on the available temporal data (time series of rainfall and evaporation in this case), so these parts of the original model did not need to be implemented within the new model.

The ICMSBuilder implementation of the model allows ready substitution of models for testing and comparison purposes. For example, a different evapotranspiration model could be added that used additional climatic data such as radiation; or a more sophisticated surface routing algorithm could be developed that distinguished between overland and channeled flow.

Tutorial 5 – Cellular automata problem

More:

■ [Conway's Game of Life](#)

Conway's Game of Life

Conway's game of life consists of a collection of "cells" which, based on some simple mathematical rules, can live, die or multiply. The life of the cells is based around the number of neighbours the cell has. A neighbour for a specific cell is defined as another populated cell existing in a cell adjacent to that cell. Therefore each cell can have up to eight (8) neighbours (left, right, up, down, and the four diagonals).

More:

- [Rules of "Life"](#)

- [Hints to implement Game of Life](#)

Rules of "Life"

The rules of Life are simple. They are dependant on whether or not a cell is already populated and the amount of neighbours that cell has.

For a cell that is populated	
Neighbours	Result
0-1	Cell dies of loneliness
2-3	Cell stays same
4+	Cell dies due to overcrowding

For a cell that is empty	
Neighbours	Result
3	Cell becomes populated

Hints to implement Game of Life

- Define the world, to be n rows by m columns.
- It is necessary for the cells to die as they reach the edge of the m by n world.
- One way of ensuring correct behaviour is to define the matrix to be $(m+2)$ by $(n+2)$, ensuring that there is a square of zeros (blanks) around the world.
- At each time step, duplicate the matrix so that the amount of neighbours is not affected by the subsequent step. Calculate the number of neighbours for a specific cell by summing up the number of adjacent matrix cells. Apply the rules. Increment time step.
- Use a map or numeric view to modify the cells, and to see the results after each time step. Try checking the continuous checkbox on the Run Manager window to make ICMS run the model continually.

Examples of the game of life on the web are:

<http://www.bitstorm.org/gameoflife/>

<http://www.mindspring.com/~alanh/life/>

<http://www.tech.org/~stuart/life/life.html>

<http://www.cs.jhu.edu/~callahan/java.html> \

You can try out your own pattern (within limits) at:

<http://www.dallas.net/~warner/life/life2.html>

Interesting patterns to try can be found at:

<http://www.cs.jhu.edu/~callahan/lexiconf.htm>

<http://home.interserv.com/~mniemiec/objname.htm>

Exercise 1. Building a project

Create and link objects using the system view

More:

- [Data](#)
- [Algorithm](#)
- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)
- [Input Data Values](#)
- [Step-By-Step Guide](#)
- [Reference Material for this Exercise](#)

Data

- areas of 10 land uses in each sub-catchment
- nutrient generation rate for each of those land uses

Algorithm

Sub-catchment load = Summation (Areas of land use * generation rate)

Approach 1

- Calculate land use areas outside ICMS and store in variables
- Store generation rates in variables
- Calculate load from each land use (by multiplying areas by generation rates) and then sum for all land uses

RESOURCES	
ICMS Project File	Exercise1.icm
Class(es)	Subcatchment
Data templates	Land Use Generation Rates (kg/ha/hr), Subcatchment Load Bushland, Established sewered urban, Unsewered peri-urban, Industrial & Commercial, Fertilised Grazing, Unfertilised Grazing, Disturbed Land, Piggery, Established unsewered urban, Built up – miscellaneous Load from bushland, Load from established sewered urban, Load from unsewered peri-urban, Load from industrial & commercial, Load from fertilised grazing, Load from unfertilised grazing, Load from disturbed land, Load from piggeries, Load from established unsewered urban, Load from built up areas
Model	Load Generation Model (Version 1)
Object	Rural Subcatchment 1

Approach 2

- Calculate land use areas and store in a matrix
- Store generation rates in a matrix
- Calculate loads from each land use and then sum for all land uses (using matrix multiplication)

Resources		
ICMS Project File	Exercise1.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 2)	
Object	Rural Subcatchment 2	

Approach 3

Represent subcatchment as a raster and record land use codes, rather than area, in each cell.

- Store land use codes in a matrix
- Calculate load from each cell (area by generation rate) and then sum for all cells

Resources		
ICMS Project File	Exercise1-3.icm	
Class(es)	Subcatchment	
Data templates	Land Use Raster (ha cells), Land Use Generation Rates (kg/ha/hr), Cell Load, Cell Area	
Model	Load Generation Model (Version 3)	
Object	Rural Subcatchment 3	

Now you realise that you actually want to compare loads from different land uses, not just the sub-catchment total.

Approach 4

While Approach 1 gives you loads from each land use, the method of storing the loads in separate variables making graphing difficult. Graphing works well with vectors.

Let’s use Approach 1 and simply move the variables to a vector in the **Finalisation** function. The vector, LULoadsVector, is then available for graphing via the Graph View.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 4)	
Object	Rural Subcatchment 4	

Approach 5

Approach 2 seems to have everything you need. Land use areas and generation rates are both stored as vectors. But their matrix multiplication results in a scalar, not a vector.

How to get a vector result? Two ways. Firstly, simply use a loop to work through the array.

Second approach - think laterally – and think of the raster as a grid. Grid multiplication is performed on a cell-by-cell basis.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 5)	
Object	Rural Subcatchment 5	

Input Data Values

Land Uses	Generation Rates* (kg/ha/yr)	Area (ha)**
Bushland	0.1	49
Established Sewered Urban	1.3	10
Unsewered Peri-Urban	0.6	0
Industrial & Commercial	1.8	0
Fertilised Grazing	1.25	15
Unfertilised Grazing	0.25	20
Disturbed Land	20.0	4
Piggery	0.3	2
Established Unsewered Urban	5.3	0
Build Up- Miscellaneous	1.8	0

* Data template name Land Use Generation Rates (kg/ha/hr) (input)

Model symbol name LandUseGenerationRates

** Data template name Land Use Areas (ha)

Model symbol name LandUseAreas

Data template CellArea contains 1

Step-By-Step Guide

Exercise 1.icm contains 1 class (with 5 models) and 5 class 'Subcatchment' objects. Each object has a different version of the 'Load Generation Model' associated with it. This design is purely to facilitate the exercise. However, it does illustrate the ease with which different models can be plugged in (and out). For each Exercise Approach, a different object is enabled (and all other objects are disabled).

More:

- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)

Approach 1

Create a sub-directory called ICMSProjects wherever you like.

Start ICMSBuilder

File|Open ExerciseProjects\Exercise1.icm

Save as ICMSProjects\Exercise1.icm

In this exercise, you are only interested in the object 'Rural Catchment 1'.

Explore the Main System View. What can you get to from here?

Right click on the object 'Rural Catchment 1' and work through the options on the speed menu.

Look at the Load Generation Model (Version 1) (by right clicking on the object on the System view and selecting Advanced|Edit model formula; or by going to the Class Information View and double clicking on the model name).

```
LoadBushland == Bushland * LandUseGenerationRates[1,1];
LoadESU = EstablishedSeweredUrban * LandUseGenerationRates[2,1];
LoadUPU = UnseweredPeriUrban * LandUseGenerationRates[3,1];
LoadIC = IndustrialCommercial * LandUseGenerationRates[4,1];
LoadFG = FertilisedGrazing * LandUseGenerationRates[5,1];
LoadUG = UnfertilisedGrazing * LandUseGenerationRates[6,1];
LoadDL = DisturbedLand * LandUseGenerationRates[7,1];
LoadPIG = Piggery * LandUseGenerationRates[8,1];
LoadEUU = EstablishedUnseweredUrban * LandUseGenerationRates[9,1];
LoadBUA = BuiltUpArea * LandUseGenerationRates[10,1];
SubcatchmentLoad = LoadBushland + LoadESU + LoadUPU + LoadIC + LoadFG
+ LoadUG + LoadDL + LoadPIG + LoadEUU + LoadBUA;
```

Look at the data templates for the class Subcatchment (*by clicking on the Class tool*). Only some of these are used in this exercise.

Let's look at the input data – 'Land Use Generation Rates (kg/ha/yr)' and one or two of the land use areas. Change the bushland generation rate to 1 (from 0.1). Look at the change in the generation rates graph and 'Subcatchment Load'.

Approach 2

In this exercise, you are only interested in the object 'Rural Catchment 2'.

We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

Let's look at the model code and the functions that are used.

```
SubcatchmentLoad = LandUseGenerationRates *  
Transpose(LandUseAreas);
```

Approach 3

In this exercise, you are only interested in the object 'Rural Catchment 3'. This is to your first introduction to the raster view and MickL program control.

We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

Let's look at the model code and the functions that are used.

```
function Initialisation()
{
// put your initialising routines here
width = MatrixWidth(LandUseRaster);
height = MatrixHeight(LandUseRaster);
CreateMatrix(CellLoad,width,height,DOUBLEPRECISION,ERASE); // to store
load for each cell
CellArea = 1; // each cell is 1 hectare
return 0;
}
```

```
function Main()
{
// put your main program here
// loop through the land use matrix and store in cell load matrix
width = MatrixWidth(LandUseRaster);
height = MatrixHeight(LandUseRaster);

for (i=1; i<=width; i++)
{
for (j = 1; j<= height; j++)
{
landusecode = LandUseRaster[i,j];
generationrate = LandUseGenerationRates[landusecode,1];
CellLoad[i,j] = CellArea * generationrate;
} // end of j loop
} // end of i loop
return 0;
}
```

Approach 4

In this exercise, you are only interested in the object 'Rural Catchment 4'. This is your first introduction to the raster view and MickL program control.

We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

```
CreateMatrix (LULoadsVector,10,1,DOUBLEPRECISION,ERASE);
```

```
LULoadsVector[1,1] = LoadBushland;
```

```
.
```

```
LULoadsVector[10,1] = LoadBUA;
```

Approach 5

In this exercise, you are interested in the object 'Rural Catchment 5'. This exercise reinforces the for statement and introduces you to the grid.

Loop solution:

```
width = MatrixWidth(LandUseGenerationRates);  
for (i=1; i<=width; i++)  
{  
  LULoadsVector[i,1] = LandUseGenerationRates[i,1] * LandUseAreas[i,1];  
} // end of for i loop
```

Grid solution:

```
LuLoadsVector = GridProd(GenerationRates, LandUseAreas, -9999);
```

Reference Material for this Exercise

- Terminology
- Classes, data templates and models
- Writing and compiling models - and good modelling practice
- Adding objects
- ICMS Functions
- Getting data into ICMS
- Viewing data in ICMS

Exercise 2. Simple nutrient load Model

Predict total load generated within a sub-catchment using a simple land use / generation rate approach

More:

- [Data](#)
- [Algorithm](#)
- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)
- [Input Data Values](#)
- [Step-By-Step Guide](#)
- [Reference Material for this Exercise](#)

Data

areas of 10 land uses in each sub-catchment
nutrient generation rate for each of those land uses

Algorithm

Sub-catchment load = Summation (Areas of land use * generation rate)

Approach 1

- Calculate land use areas outside ICMS and store in variables
- Store generation rates in variables
- Calculate load from each land use (by multiplying areas by generation rates) and then sum for all land uses

RESOURCES	
ICMS Project File	Exercise1.icm
Class(es)	Subcatchment
Data templates	Land Use Generation Rates (kg/ha/hr), Subcatchment Load Bushland, Established sewer urban, Unsewered peri-urban, Industrial & Commercial, Fertilised Grazing, Unfertilised Grazing, Disturbed Land, Piggery, Established unsewered urban, Built up – miscellaneous Load from bushland, Load from established sewer urban, Load from unsewered peri-urban, Load from industrial & commercial, Load from fertilised grazing, Load from unfertilised grazing, Load from disturbed land, Load from piggeries, Load from established unsewered urban, Load from built up areas
Model	Load Generation Model (Version 1)
Object	Rural Subcatchment 1

Approach 2

Calculate land use areas and store in a matrix

Store generation rates in a matrix

Calculate loads from each land use and then sum for all land uses (using matrix multiplication)

Resources		
ICMS Project File	Exercise1.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 2)	
Object	Rural Subcatchment 2	

Approach 3

Represent subcatchment as a raster and record land use codes, rather than area, in each cell.

Store land use codes in a matrix

Calculate load from each cell (area by generation rate) and then sum for all cells

Resources		
ICMS Project File	Exercise1-3.icm	
Class(es)	Subcatchment	
Data templates	Land Use Raster (ha cells), Land Use Generation Rates (kg/ha/hr), Cell Load, Cell Area	
Model	Load Generation Model (Version 3)	
Object	Rural Subcatchment 3	

Now you realise that you actually want to compare loads from different land uses, not just the sub-catchment total.

Approach 4

While Approach 1 gives you loads from each land use, the method of storing the loads in separate variables making graphing difficult. Graphing works well with vectors.

Let’s use Approach 1 and simply move the variables to a vector in the **Finalisation** function. The vector, LULoadsVector, is then available for graphing via the Graph View.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 4)	
Object	Rural Subcatchment 4	

Approach 5

Approach 2 seems to have everything you need. Land use areas and generation rates are both stored as vectors. But their matrix multiplication results in a scalar, not a vector.

How to get a vector result? Twp ways. Firstly, simply use a loop to work through the array.

Second approach - think laterally – and think of the raster as a grid. Grid multiplication is performed on a cell-by-cell basis.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 5)	
Object	Rural Subcatchment 5	

Input Data Values

Land Uses	Generation Rates* (kg/ha/yr)	Area (ha)**
Bushland	0.1	49
Established Sewered Urban	1.3	10
Unsewered Peri-Urban	0.6	0
Industrial & Commercial	1.8	0
Fertilised Grazing	1.25	15
Unfertilised Grazing	0.25	20
Disturbed Land	20.0	4
Piggery	0.3	2
Established Unsewered Urban	5.3	0
Build Up- Miscellaneous	1.8	0

* Data template name Land Use Generation Rates (kg/ha/hr) (input)
Model symbol name LandUseGenerationRates
** Data template name Land Use Areas (ha)
Model symbol name LandUseAreas

Data template CellArea contains 1

Step-By-Step Guide

Exercise 1.icm contains 1 class (with 5 models) and 5 class 'Subcatchment' objects. Each object has a different version of the 'Load Generation Model' associated with it. This design is purely to facilitate the exercise. However, it does illustrate the ease with which different models can be plugged in (and out). For each Exercise Approach, a different object is enabled (and all other objects are disabled).

More:

- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)

Approach 1

Create a sub-directory called ICMSProjects wherever you like.

Start ICMSBuilder

File|Open ExerciseProjects\Exercise1.icm

Save as ICMSProjects\Exercise1.icm

In this exercise, you are only interested in the object 'Rural Catchment 1'.

Explore the Main System View. What can you get to from here?

Right click on the object 'Rural Catchment 1' and work through the options on the speed menu.

Look at the Load Generation Model (Version 1) (by right clicking on the object on the System view and selecting Advanced|Edit model formula; or by going to the Class Information View and double clicking on the model name).

```
LoadBushland == Bushland * LandUseGenerationRates[1,1];  
LoadESU = EstablishedSeweredUrban *  
LandUseGenerationRates[2,1];  
LoadUPU = UnseweredPeriUrban * LandUseGenerationRates[3,1];  
LoadIC = IndustrialCommercial * LandUseGenerationRates[4,1];  
LoadFG = FertilisedGrazing * LandUseGenerationRates[5,1];  
LoadUG = UnfertilisedGrazing * LandUseGenerationRates[6,1];  
LoadDL = DisturbedLand * LandUseGenerationRates[7,1];  
LoadPIG = Piggery * LandUseGenerationRates[8,1];  
LoadEUU = EstablishedUnseweredUrban *  
LandUseGenerationRates[9,1];  
LoadBUA = BuiltUpArea * LandUseGenerationRates[10,1];  
SubcatchmentLoad = LoadBushland + LoadESU + LoadUPU +  
LoadIC + LoadFG + LoadUG + LoadDL + LoadPIG + LoadEUU +  
LoadBUA;
```

Look at the data templates for the class Subcatchment (*by clicking on the Class tool*). Only some of these are used in this exercise.

Let's look at the input data – 'Land Use Generation Rates (kg/ha/yr)' and one or two of the land use areas. Change the bushland generation rate to 1 (from 0.1). Look at the change in the generation rates graph and 'Subcatchment Load'.

Approach 2

In this exercise, you are only interested in the object 'Rural Catchment 2'. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

Let's look at the model code and the functions that are used.

```
SubcatchmentLoad = LandUseGenerationRates *  
Transpose(LandUseAreas);
```

Approach 3

In this exercise, you are only interested in the object 'Rural Catchment 3'. This is to your first introduction to the raster view and MickL program control. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance. Let's look at the model code and the functions that are used.

function Initialisation()

```
{  
// put your initialising routines here  
width = MatrixWidth(LandUseRaster);  
height = MatrixHeight(LandUseRaster);  
CreateMatrix(CellLoad,width,height,DOUBLEPRECISION,ERASE);  
// to store load for each cell  
CellArea = 1; // each cell is 1 hectare  
return 0;  
}
```

function Main()

```
{  
// put your main program here  
// loop through the land use matrix and store in cell load matrix  
width = MatrixWidth(LandUseRaster);  
height = MatrixHeight(LandUseRaster);  
  
for (i=1; i<=width; i++)  
{  
  for (j = 1; j<= height; j++)  
  {  
    landusecode = LandUseRaster[i,j];  
    generationrate = LandUseGenerationRates[landusecode,1];  
    CellLoad[i,j] = CellArea * generationrate;  
  } // end of j loop  
} // end of i loop  
return 0;
```

Approach 4

In this exercise, you are only interested in the object 'Rural Catchment 4'. This is your first introduction to the raster view and MickL program control. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

```
CreateMatrix (LULoadsVector,10,1,DOUBLEPRECISION,ERASE);
```

```
LULoadsVector[1,1] = LoadBushland;
```

```
.
```

```
LULoadsVector[10,1] = LoadBUA;
```

Approach 5

In this exercise, you are interested in the object 'Rural Catchment 5'. This exercise reinforces the for statement and introduces you to the grid.

Loop solution:

```
width = MatrixWidth(LandUseGenerationRates);  
for (i=1; i<=width; i++)  
{  
LULoadsVector[i,1] = LandUseGenerationRates[i,1] *  
LandUseAreas[i,1];  
} // end of for i loop
```

Grid solution:

```
LuLoadsVector = GridProd(GenerationRates, LandUseAreas, -9999);
```

Reference Material for this Exercise

Terminology

Classes, data templates and models

Writing and compiling models - and good modelling practice

Adding objects

ICMS Functions

Getting data into ICMS

Viewing data in ICMS

Exercise 3. Routing material through a catchment

Predict total load generated within a sub-catchment using a simple land use / generation rate approach, and then route it through a network of subcatchments

More:

- [Data](#)
- [Algorithm](#)
- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)
- [Input Data Values](#)
- [Step-By-Step Guide](#)
- [Reference Material for this Exercise](#)

Data

areas of 10 land uses in each sub-catchment
nutrient generation rate for each of those land uses

Algorithm

Sub-catchment load = Summation (Areas of land use * generation rate)

Approach 1

- Calculate land use areas outside ICMS and store in variables
- Store generation rates in variables
- Calculate load from each land use (by multiplying areas by generation rates) and then sum for all land uses

RESOURCES	
ICMS Project File	Exercise1.icm
Class(es)	Subcatchment
Data templates	Land Use Generation Rates (kg/ha/hr), Subcatchment Load Bushland, Established sewer urban, Unsewered peri-urban, Industrial & Commercial, Fertilised Grazing, Unfertilised Grazing, Disturbed Land, Piggery, Established unsewered urban, Built up – miscellaneous Load from bushland, Load from established sewer urban, Load from unsewered peri-urban, Load from industrial & commercial, Load from fertilised grazing, Load from unfertilised grazing, Load from disturbed land, Load from piggeries, Load from established unsewered urban, Load from built up areas
Model	Load Generation Model (Version 1)
Object	Rural Subcatchment 1

Approach 2

Calculate land use areas and store in a matrix

Store generation rates in a matrix

Calculate loads from each land use and then sum for all land uses (using matrix multiplication)

Resources		
ICMS Project File	Exercise1.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 2)	
Object	Rural Subcatchment 2	

Approach 3

Represent subcatchment as a raster and record land use codes, rather than area, in each cell.

Store land use codes in a matrix

Calculate load from each cell (area by generation rate) and then sum for all cells

Resources		
ICMS Project File	Exercise1-3.icm	
Class(es)	Subcatchment	
Data templates	Land Use Raster (ha cells), Land Use Generation Rates (kg/ha/hr), Cell Load, Cell Area	
Model	Load Generation Model (Version 3)	
Object	Rural Subcatchment 3	

Now you realise that you actually want to compare loads from different land uses, not just the sub-catchment total.

Approach 4

While Approach 1 gives you loads from each land use, the method of storing the loads in separate variables making graphing difficult. Graphing works well with vectors.

Let’s use Approach 1 and simply move the variables to a vector in the **Finalisation** function. The vector, LULoadsVector, is then available for graphing via the Graph View.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 4)	
Object	Rural Subcatchment 4	

Approach 5

Approach 2 seems to have everything you need. Land use areas and generation rates are both stored as vectors. But their matrix multiplication results in a scalar, not a vector.

How to get a vector result? Twp ways. Firstly, simply use a loop to work through the array.

Second approach - think laterally – and think of the raster as a grid. Grid multiplication is performed on a cell-by-cell basis.

Resources		
ICMS Project File	Exercise1-4&5.icm	
Class(es)	Subcatchment	
Data templates	Land Use Areas (ha), Land Use Generation Rates (kg/ha/hr), Subcatchment Load	
Model	Load Generation Model (Version 5)	
Object	Rural Subcatchment 5	

Input Data Values

Land Uses	Generation Rates* (kg/ha/yr)	Area (ha)**
Bushland	0.1	49
Established Sewered Urban	1.3	10
Unsewered Peri-Urban	0.6	0
Industrial & Commercial	1.8	0
Fertilised Grazing	1.25	15
Unfertilised Grazing	0.25	20
Disturbed Land	20.0	4
Piggery	0.3	2
Established Unsewered Urban	5.3	0
Build Up- Miscellaneous	1.8	0

* Data template name Land Use Generation Rates (kg/ha/hr) (input)
Model symbol name LandUseGenerationRates
** Data template name Land Use Areas (ha)
Model symbol name LandUseAreas

Data template CellArea contains 1

Step-By-Step Guide

Exercise 1.icm contains 1 class (with 5 models) and 5 class 'Subcatchment' objects. Each object has a different version of the 'Load Generation Model' associated with it. This design is purely to facilitate the exercise. However, it does illustrate the ease with which different models can be plugged in (and out). For each Exercise Approach, a different object is enabled (and all other objects are disabled).

More:

- [Approach 1](#)
- [Approach 2](#)
- [Approach 3](#)
- [Approach 4](#)
- [Approach 5](#)

Approach 1

Create a sub-directory called ICMSProjects wherever you like.

Start ICMSBuilder

File|Open Exercises\Exercise1.icm

Save as ICMSProjects\Exercise1.icm

In this exercise, you are only interested in the object 'Rural Catchment 1'.

Explore the Main System View. What can you get to from here?

Right click on the object 'Rural Catchment 1' and work through the options on the speed menu.

Look at the Load Generation Model (Version 1) (by right clicking on the object on the System view and selecting Advanced|Edit model formula; or by going to the Class Information View and double clicking on the model name).

```
LoadBushland == Bushland * LandUseGenerationRates[1,1];  
LoadESU = EstablishedSeweredUrban *  
LandUseGenerationRates[2,1];  
LoadUPU = UnseweredPeriUrban * LandUseGenerationRates[3,1];  
LoadIC = IndustrialCommercial * LandUseGenerationRates[4,1];  
LoadFG = FertilisedGrazing * LandUseGenerationRates[5,1];  
LoadUG = UnfertilisedGrazing * LandUseGenerationRates[6,1];  
LoadDL = DisturbedLand * LandUseGenerationRates[7,1];  
LoadPIG = Piggery * LandUseGenerationRates[8,1];  
LoadEUU = EstablishedUnseweredUrban *  
LandUseGenerationRates[9,1];  
LoadBUA = BuiltUpArea * LandUseGenerationRates[10,1];  
SubcatchmentLoad = LoadBushland + LoadESU + LoadUPU +  
LoadIC + LoadFG + LoadUG + LoadDL + LoadPIG + LoadEUU +  
LoadBUA;
```

Look at the data templates for the class Subcatchment (*by clicking on the Class tool*). Only some of these are used in this exercise.

Let's look at the input data – 'Land Use Generation Rates (kg/ha/yr)' and one or two of the land use areas. Change the bushland generation rate to 1 (from 0.1). Look at the change in the generation rates graph and 'Subcatchment Load'.

Approach 2

In this exercise, you are only interested in the object 'Rural Catchment 2'. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

Let's look at the model code and the functions that are used.

```
SubcatchmentLoad = LandUseGenerationRates *  
Transpose(LandUseAreas);
```

Approach 3

In this exercise, you are only interested in the object 'Rural Catchment 3'. This is to your first introduction to the raster view and MickL program control. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance. Let's look at the model code and the functions that are used.

function Initialisation()

```
{  
// put your initialising routines here  
width = MatrixWidth(LandUseRaster);  
height = MatrixHeight(LandUseRaster);  
CreateMatrix(CellLoad,width,height,DOUBLEPRECISION,ERASE); //  
to store load for each cell  
CellArea = 1; // each cell is 1 hectare  
return 0;  
}
```

function Main()

```
{  
// put your main program here  
// loop through the land use matrix and store in cell load matrix  
width = MatrixWidth(LandUseRaster);  
height = MatrixHeight(LandUseRaster);  
  
for (i=1; i<=width; i++)  
{  
for (j = 1; j<= height; j++)  
{  
landusecode = LandUseRaster[i,j];  
generationrate = LandUseGenerationRates[landusecode,1];  
CellLoad[i,j] = CellArea * generationrate;  
} // end of j loop  
} // end of i loop  
return 0;  
}
```

Approach 4

In this exercise, you are only interested in the object 'Rural Catchment 4'. This is your first introduction to the raster view and MickL program control. We will work with the 'Land Use Area (ha)' matrix. Create a numeric view and graph view of this data instance.

```
CreateMatrix (LULoadsVector,10,1,DOUBLEPRECISION,ERASE);
```

```
LULoadsVector[1,1] = LoadBushland;
```

```
.
```

```
LULoadsVector[10,1] = LoadBUA;
```

Approach 5

In this exercise, you are interested in the object 'Rural Catchment 5'. This exercise reinforces the for statement and introduces you to the grid.

Loop solution:

```
width = MatrixWidth(LandUseGenerationRates);  
for (i=1; i<=width; i++)  
{  
LULoadsVector[i,1] = LandUseGenerationRates[i,1] *  
LandUseAreas[i,1];  
} // end of for i loop
```

Grid solution:

```
LuLoadsVector = GridProd(GenerationRates, LandUseAreas, -9999);
```

Reference Material for this Exercise

Terminology

Classes, data templates and models

Writing and compiling models - and good modelling practice

Adding objects

ICMS Functions

Getting data into ICMS

Viewing data in ICMS