Object Oriented Programming With Omnis Studio

Richard Mortimer
Technical Consultant
richard.mortimer@omnis.net

Topics

- What is Object Orientation?
- The Principles of Object Orientation
- Object Orientation in Omnis Studio
- Using Inheritance and Subclasses
- Invoking methods and messaging
- Using Subwindows
- Using Object Classes

Omnis

Structured Programming

- Top-down approach
- Data and functions are kept separate
- Breaks a program down into components until the components cannot be decomposed anymore
- Improved the quality of software
- If design is found to be incorrect after programming has started, then the design may have to be entirely restructured

Omnis

What Is Object Orientation?

- Application based on software objects
- These simulate real-world objects
- It is a different way to construct systems
- It is a paradigm shift
- It is a technology
- More importantly it is a methodology

Omnis

A Bit of OO History

- Invented in the late 1960s in a language called Simula by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Centre in Oslo
- Early 1970s, SmallTalk by Alan Kay at Xerox PARC, furthered the idea of using software objects simulating real-world objects to using software objects for prototyping and developing applications
- Mid 1980s, other OO programming languages emerged, such as C++ and Eiffel
- Late 1990s, Java developed for the web and Omnis Studio

Omnis

Objects

- Objects are "black boxes" that communicate with each other to perform tasks
- Objects combine both ''state'' (i.e. data) and ''behavior'' (i.e. procedures or ''methods'') into a single entity
- This speeds the development of new programs, and improves
  - Consistency
  - Maintenance
  - Reusability

Omnis

Principles of Object Orientation

- Abstraction
- Classes
  - Encapsulation
- Instances
- Messaging
  - Polymorphism
- Inheritance

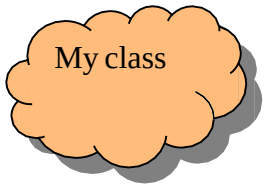
Omnis

Abstraction

- Abstraction is the ability of a language to take a concept and create an abstract representation of it within a program
- It involves identifying or abstracting common features of objects and procedures and then combining them into a single entity that can represent them
- For example, a programmer would use abstraction to note that two functions perform almost the same task and can be combined into a single function
- Each object in the system serves as a model of an abstract "actor" that can perform work, report on and change its state, and "communicate" with other objects in the system
- A Customer object, for instance, is an abstract representation of a real-world customer

Classes

- You can think of a class as a factory that can produce just one kind of object
- An object is defined by its class, which determines everything about an object
- Classes provide the specifications for the objects' behaviors and attributes
- Class is an abstraction of a real-world entity



Omnis

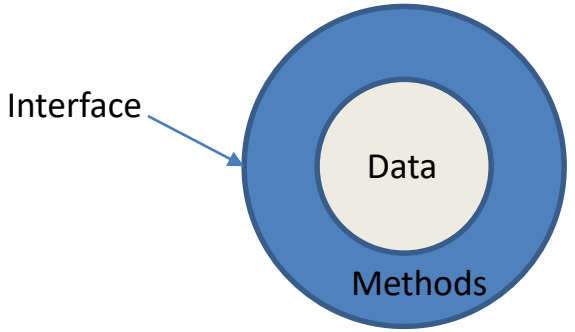
Instances

- Objects are individual instances of a class
- Each instance has a unique identifier
- For example, you may create an object called Spot from the class Dog
- Object-oriented languages have some means, usually called a factory, to "manufacture" object instances from a class definition
- You could make more than one object from the Dog class, and call them Spot, Rover, Wellard, etc.
- Instances communicate with each other using Messaging

Omnis

Encapsulation

- We can think of an object as having an external interface and an internal environment that is hidden from the outside world
- This information hiding is known as encapsulation
- Artefacts that are hidden
  - Data (variables)
  - Private methods
- The hidden data is protected
- Artefacts that are not hidden
  - Public methods
- The internals may be modified without affecting the outside world

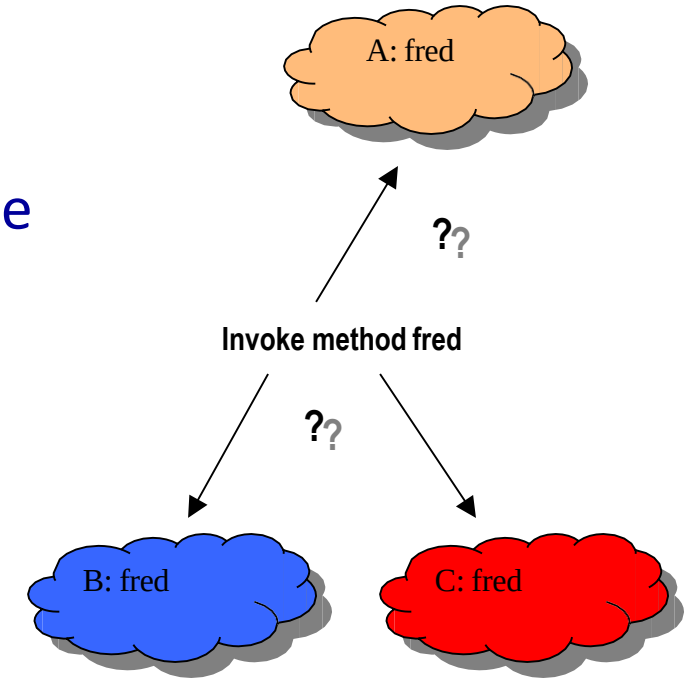


Messaging

- Instances communicate with each other via messaging
- The valid messages are defined in the class
- When an instance receives a messages that it understands, it performs an operation
- For example, the Dog class defines what it is to be a Dog object, so that the Dog objects understands, and can act upon messages such as "bark", "fetch", and "roll-over“
- The message contains
  - The name of the operation (Method name)
  - Any additional information that the operation requires (Parameters)
- An operation may return data
  - This is a way to access encapsulated data

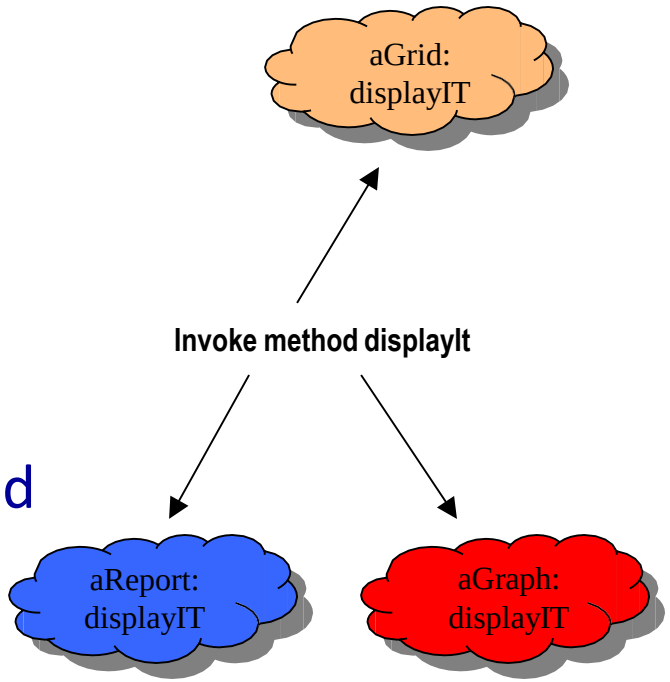
Late Binding

- Traditional languages use Static binding to bind a reference to a particular variable or method at design (compile) time
- With Dynamic or Late binding the decision as to which variable or method to use is postponed to runtime
- It is often impossible at design time to say with any degree of certainty what variable or method will actually be used



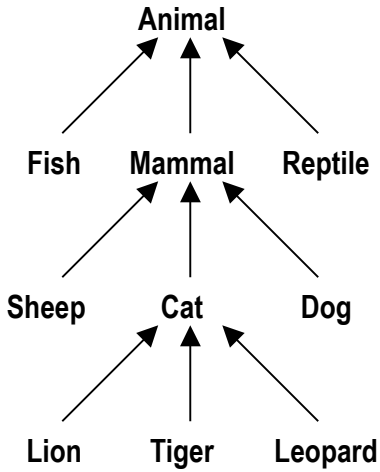
Polymorphism

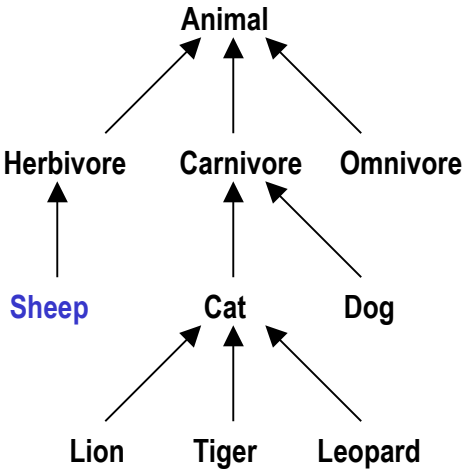
- Each class of object that responds to a message has its own implementation of the method with a common method name
- When the method is called the object responds with its own implementation of the method
- The invoking object doesn't need to know what kind of object is receiving the message



Inheritance

- The ability to define a class as a specialization of another class
- Inheritance is hierarchical
- Subclass inherits properties from super class
- Sometimes terms derived class and base class used instead of subclass & super class
- Relationship between subclass and super class “is a kind of”
- For example, a Tiger is a kind of Cat
- Inheritance hierarchies can be expressed in different ways





Software Class Inheritance

- What if there is already a class that can respond to a number of different messages?
- What if you wanted to make a new, similar class which adds just a couple more messages?
- Why rewrite the entire class?
- Inheritance provides a simple and elegant way to reuse code and to model the real world in a meaningful way
- Some additional methods may be defined to extend the capabilities of the class

Omnis

What is Inherited?

- Visual attributes
- Certain variables
- External interface
  - Public methods

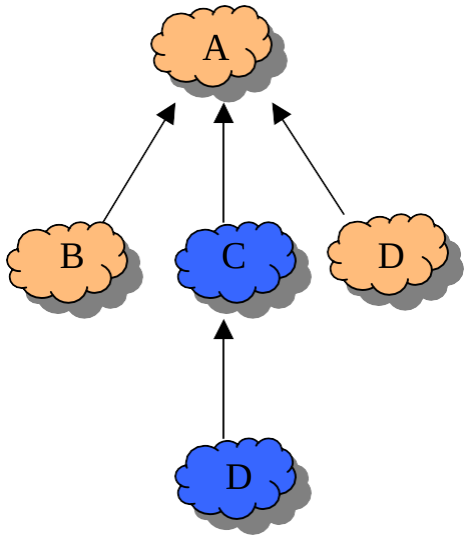
Omnis

Overriding Properties and Methods

- To create a subclass is *specialization*
- To combine common parts of derived classes into a common base (or parent) is *generalization*
- *Overriding* is the term used in OO languages for redefining a method in a derived class, thus providing specialized behavior
- The property or method in the subclass replaces the inherited one

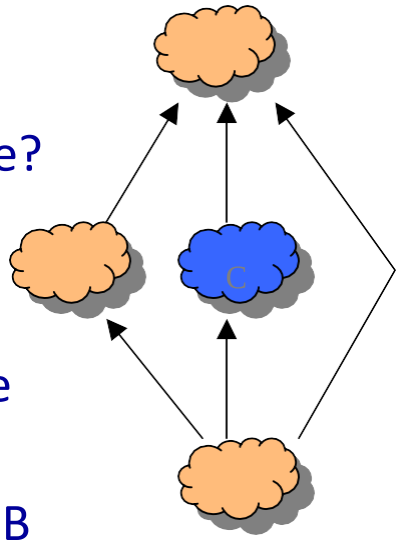
Single Inheritance

- In Single Inheritance a child subclass can only inherit from a single parent super class
- Any super class can have multiple subclasses
- Simple and clean mechanism



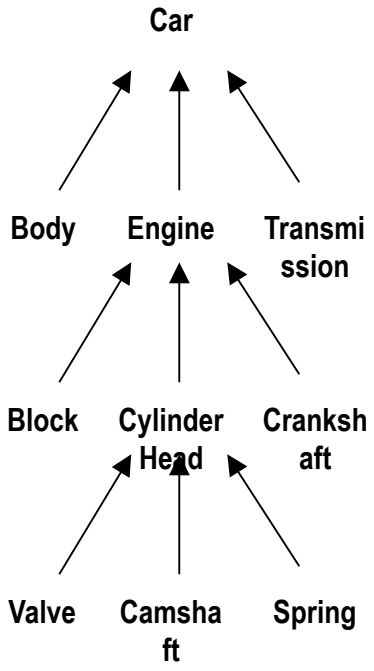
Multiple Inheritance

- Multiple Inheritance occurs when a class inherits from more than one parent super class
- What difficulties are posed by Multiple Inheritance?
- A. Produces paradoxes through repeated inheritance of properties, e.g.
  - 2 classes B and C inherit from class A and there is another class D that is inheriting A, B and C
  - Now if class A has a method fred() which class B and class C both inherit then class D will also inherit it from A, B and C
  - When fred() in D is called, which fred should be called, i.e. A's, B's or C's?



Aggregation and Containers

- An alternative hierarchy to inheritance in which components are collected together
- Not inheritance (an engine is not a special kind of car)
- Objects in an aggregation are close-coupled, they will only 'work' in the aggregation
- Containers are like shopping baskets that can hold many different kinds of objects
- Objects in a container are loosely coupled, each object is independently useful



Omnis

What is an Object Oriented Language?

- In 1987 Peter Wenger proposed a definition for object-oriented languages
- For a programming language to be object oriented he required that it:
  - Be object based, meaning that you can easily make encapsulated program objects in it
  - Be class based, meaning that every object belongs to or is manufactured from a class
  - Support inheritance, meaning that classes may be arranged in a super class-subclass hierarchy

Object Orientation in Omnis Studio

- Now we are going to look at the features of Omnis Studio that enable developers to build Object Oriented applications
  - Classes
  - Inheritance
  - Instances
  - Messaging

Omnis Studio Classes

- GUI classes
  - Window*, menu*, toolbar*, remote form*, report*
- Data classes
  - Schema†, query†, table*, file, search
- Non visual logic classes
  - Object*, task*, remote task*, code

* OO classes that require instantiation
† Schema and query instantiated via table class

What is Inherited by a Subclass?

- Omnis Studio only provides for single inheritance
- Public methods
- Class variables
- Instance variables
- Properties
- Subordinate objects
  - Fields on window classes
  - Tools on toolbars
  - Lines on menu classes

Omnis

Public and Private Methods

- Public methods are prefixed by a $ sign
- Other methods are Private
- Invoke a private method using the Do method command e.g.

```
Do method fred(p1,p2,p3)
```

  - Parameters are supplied in the parentheses
  - This is not a message based call
- Do not use the Do method command to call public methods even if they are in the same class
- Public methods should be called using a message

Instantiating a Class

- Using a 4GL command
  ```
  Open window instance myWind/Cen (p1,p2)
  ```
- Using a notation command
  ```
  Do $windows.myWind.$open('*',kWindowCenter,p1,p2)
  ```
- Command refers to the class name
- 1st parameter is name of instance or '*' to generate instance name
- $construct method runs automatically when an instance is created

Other Examples of Instantiation

- Cascade menu instantiated at same time as parent menu
- Menu on window menu bar instantiated at same time as parent window
- Context menu instantiated when user right-clicks on parent object (window or field)
- Report instance
- Using a 4GL commands

```
Set report name myRep
Prepare for print {* (#1,#2)}
```

- Using a notation command

```
Do $reports.myRep.$open('*',p1,p2)
```

- Note that the following command also instantiates a report but behaves differently

```
Print report {* (p1,p2)}
```

Instance Groups

- There are group notation items for instantiated classes
- These contain a single member for each instance
  - `Open window instances - $iwindows`
  - Task instances - $itasks
    - Name is the same as the library name
  - Installed menu instances - $imenus
    - Only menus installed on main Omnis menu bar
  - Installed toolbar instances - $itoolbars
    - Only menus installed on main Omnis toolbar
- Window instances have groups for objects installed on them
  - `$iwindows.myWind.$menus.myMenu`
  - `$iwindows.myWind.$toolbars.myToolbar`

Destroying an Instance

- Using a 4GL command
  ```
  Close window instance myWind
  ```
- Using a notation command
  ```
  Do $iwindows.myWind.$close()
  ```
- Command refers to the Instance name
- $destruct method runs automatically when an instance is destroyed except for Table and Object
  - For Table and Object instances you can run it manually from the parent instance (e.g. in window $destruct method)
    ```
    Do myRow.$destruct()
    ```

Sending a Message

- You can send a message to an object using the Do command
- A message can only be sent to an Instance
- You must have a reference (like a “handle”) to the object that you are sending the message to

```
Do $iwindows.myWind.$fred(p1,p2,p3)
```

  - This is like the address on a letter
  - Parameters are supplied in the parentheses
  - $iwindows.myWind resolves to a reference
- You cannot send a message to a context menu instance since it is not accessible via group notation

Broadcasting a Message

- You can send the same message to a number of objects in a group using the $sendall method

```
Do $iwindows.$sendall($ref.$close())
```

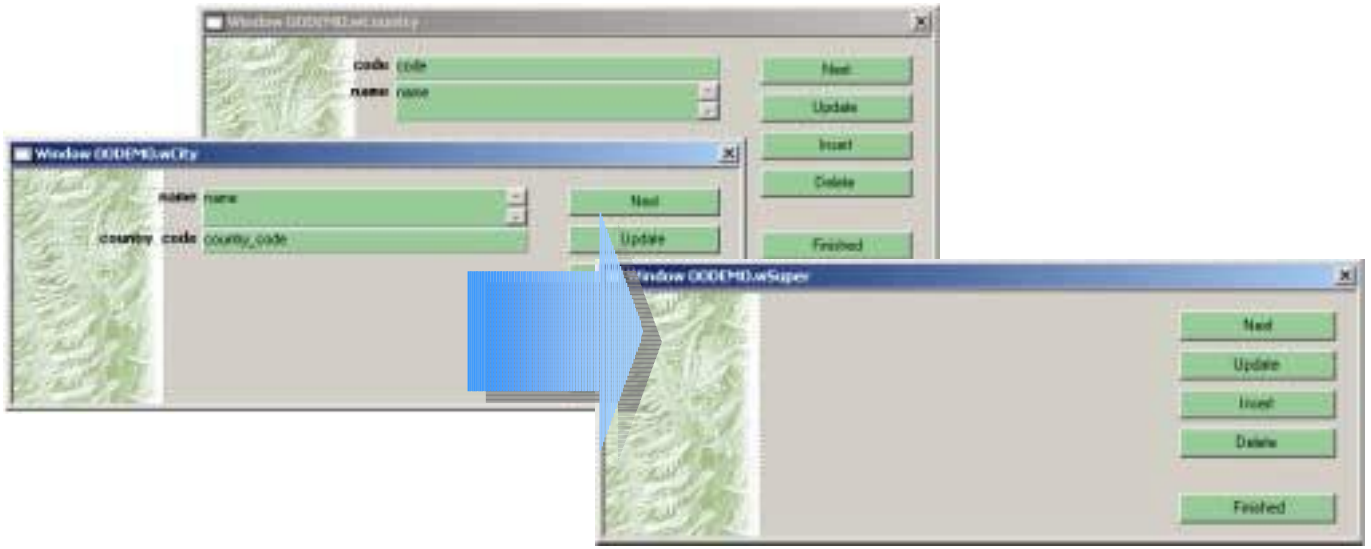
  - The method iterates all the member objects of a group
  - For each iteration $ref references the current object
- The method has a second optional Boolean parameter so that you can be selective

```
Do $iwindows.$sendall($ref.$close(),
$ref.$class().$name='myClass')
```

- The message will only be sent to the object if the 2nd parameter evaluates to kTrue or a non-zero value

Creating a Super Class

- Identify common features and behavior
- Create an abstraction containing common features
- This becomes the basis for your Super Class

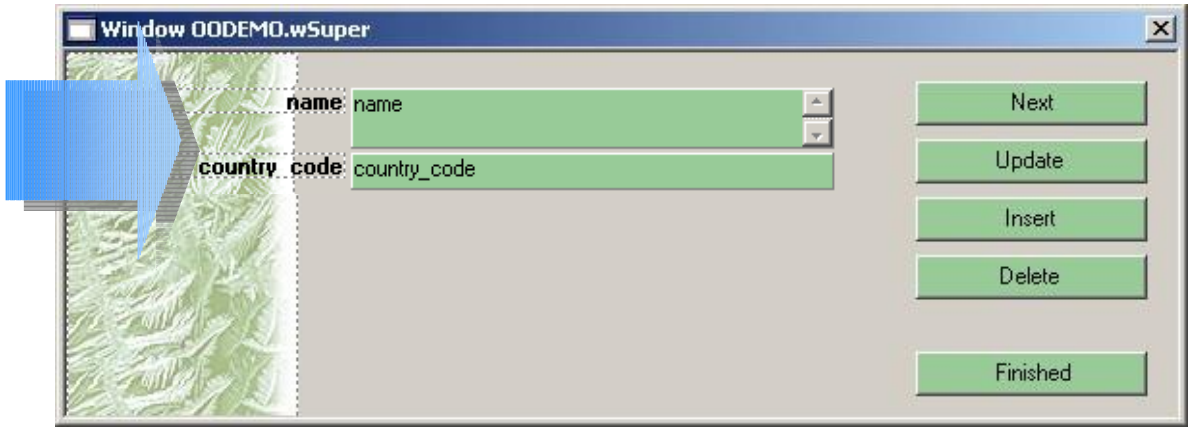


Creating a Subclass

- Right-click the Super Class in the Browser window
- Select Make Subclass
- Give the Subclass an appropriate name
- Note that the Super Class is displayed in the Browser Window

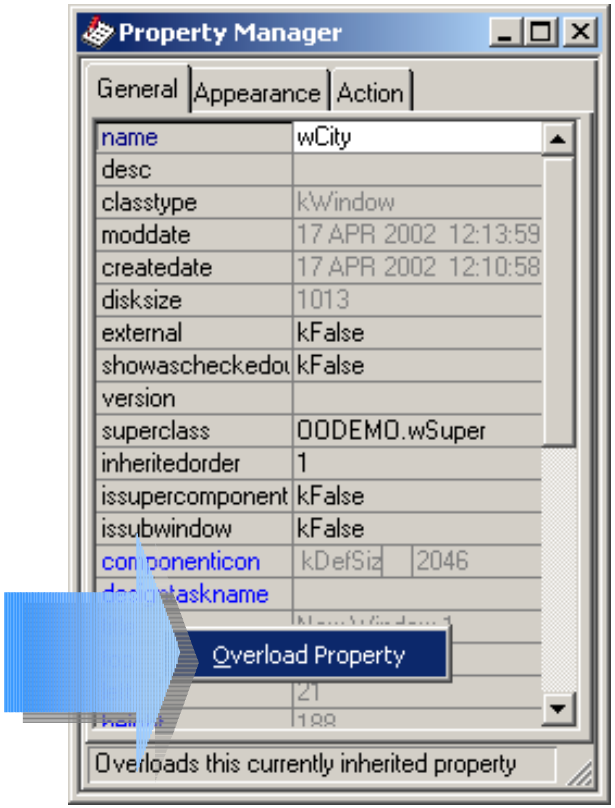
Make the Subclass a Specialization

- Add the features that make the Subclass a specialization of the Super Class
- Add fields
- Override properties and methods



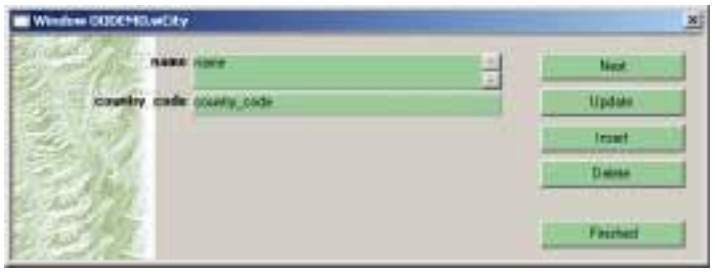
Overloading properties

- You can override or “overload” many inherited properties using the Property Manager window
- Inherited properties are shown in blue
- Select the object in the Window Editor
- Right-click the required property and select Overload Property
- Enter the new value on the right as normal



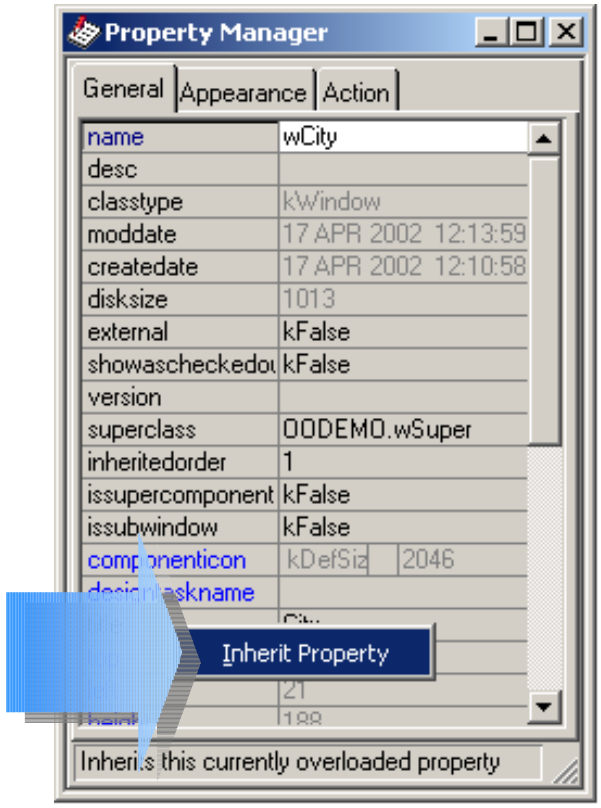
What You Cannot Overload

- You cannot overload properties of objects contained in an inherited class
- E.g. the position properties top, left, height and width of our pushbutton fields cannot be overloaded
- Hint: Use floating field properties to reposition fields automatically when a subclass window is resized or you can use notation to modify them in
  - $construct method



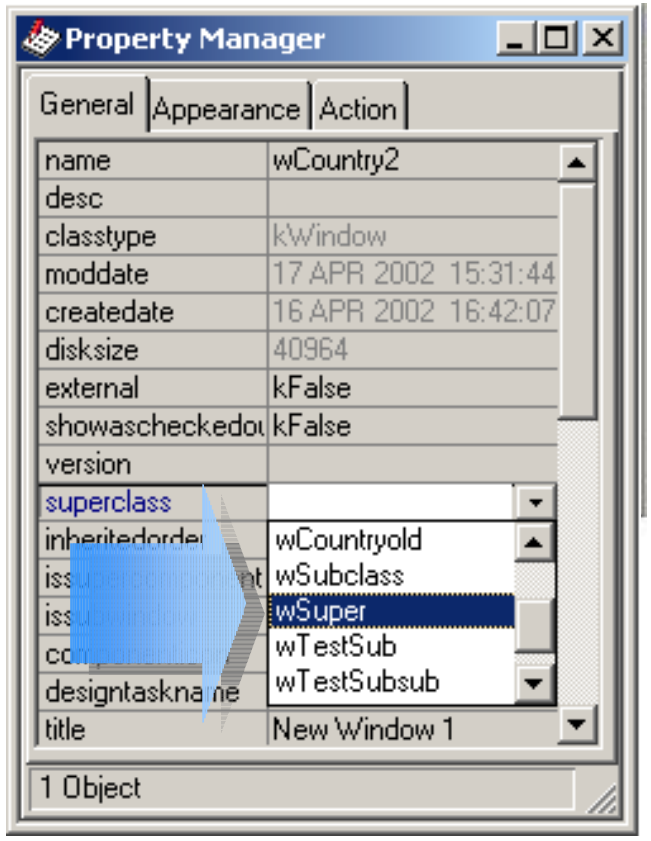
Inheriting properties

- If a property has been previously overloaded then it can be inherited from the Super Class using the Property Manager window
- Non-Inherited properties are shown in black
- Select the object in the Window Editor
- Right-click the property and select Inherit Property
- The inherited value will be shown on the right in blue



Setting the superclass Property

- An alternative way of establishing a Subclass is to set its $superclass property
- Right-click the class in the Browser and select Properties
- Select the required Super Class from the list
- Not recommended because common properties are not automatically inherited (i.e. they must be manually inherited)

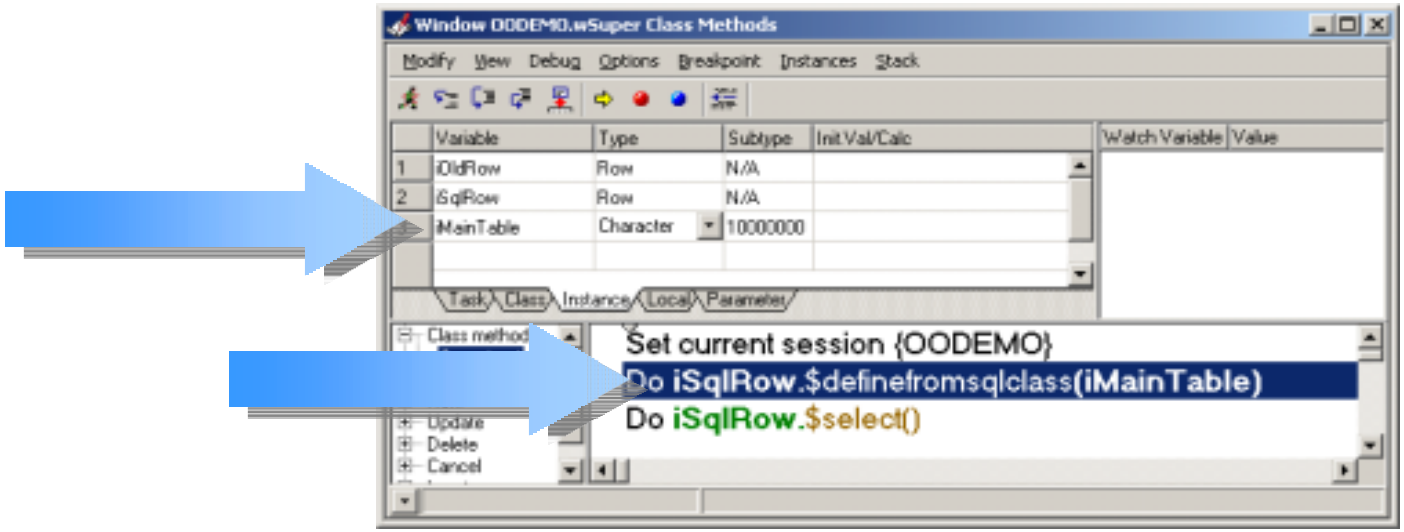


Inherited methods

- Only Public methods prefixed by a $ sign are inherited
- Inherited methods are shown in blue
- When the method is invoked by a message, the method in the Super Class is executed

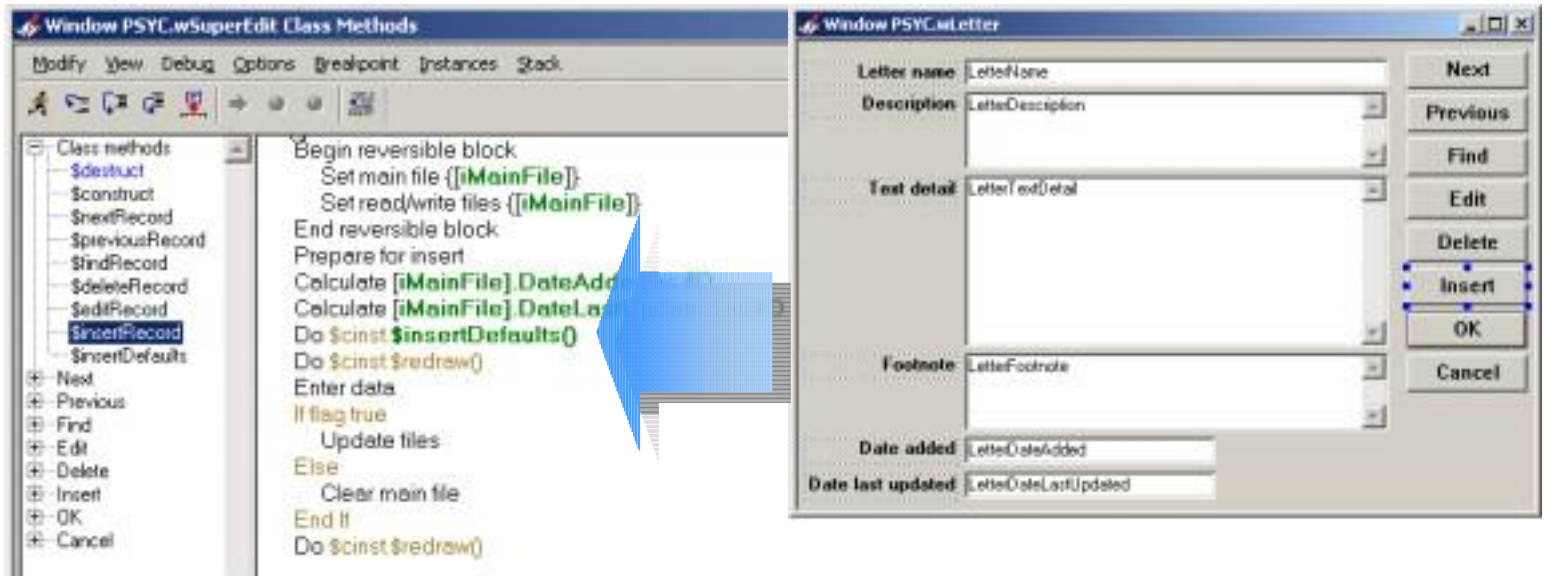
Ensure Methods are Abstracted

- Super Class methods must be generic abstractions
- Code referring to specific objects should be removed to a Subclass and the method overridden in the Subclass
- Alternatively, add an instance variable to the Super Class so that the specific values can be communicated from the Subclass



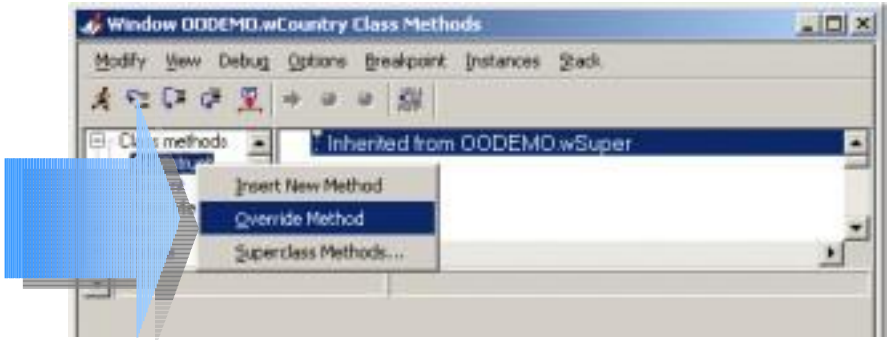
Callout Stubs

- Sometimes there is a requirement to call out to a subclass method during execution of a Super Class method
- Create a dummy public method containing no code, defined in Super Class and invoked by other Super Class methods
- Override in Subclass if required



Overriding methods

- You can override an inherited method in a Subclass
- Right-click the required method in the Method Editor and select Override Method
- Enter the new code to the right as normal
- When the method is invoked by a message, the method in the Subclass is executed



Invoking the Super Class Methods

- You can invoke the Super Class version of a method that was overridden using the command

  ```
  Do inherited
  ```

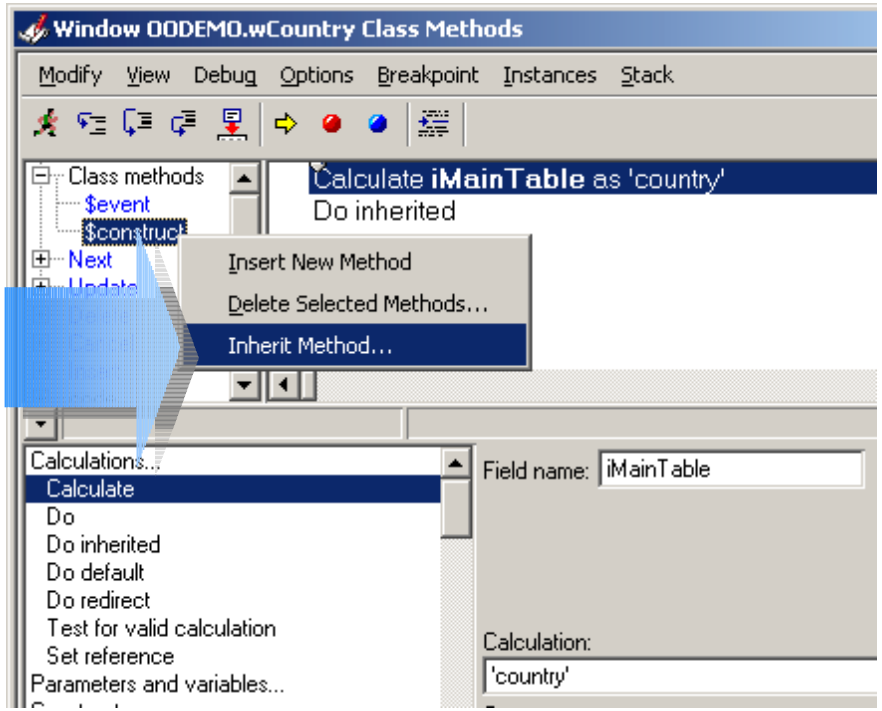
  - Parameters are supplied to both methods as normal
  - You cannot pass different parameters using this technique
  - However both methods may use Field Reference parameters to refer to the same data

- If you need to override the parameters that would normally be sent to the Super Class method then use

  ```
  Do $inherited.$myMethod(p7,p8,p9)
  ```

Inheriting Methods

- A method that has been overridden in a Subclass can be inherited
- Right-click in the Method Editor and select Inherit Method
- Note that to do this the Subclass method is removed and all code lines are lost



Static vs. Dynamic Variable Refs

- Static reference

```
Calculate iCust as 'Fred'
Calculate myVar as iCust
```

  - Reference to iCust is tokenized
- Dynamic reference

```
Calculate $cinst.iCust as 'Jill'
Calculate myVar as $cinst.iCust
```

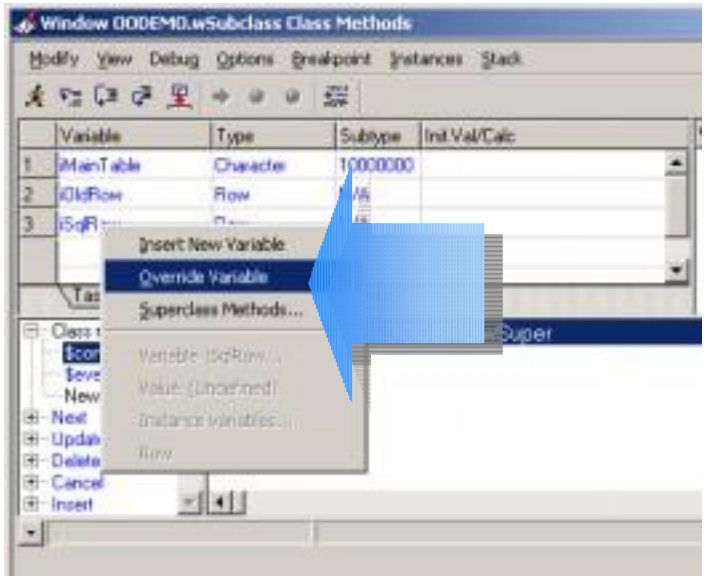
  - Nb. iCust is simply text resolved to the variable at runtime
- Watch your spelling with dynamic references (unlike static references they are not checked by the IDE)

```
Calculate $cinst.iCus as 'Sam'
Calculate myVar as $cinst.iCus
```

  - In this case myVar will contain a Null value

Overriding Variables

- Inherited variables shown in blue
- Right-click on variable in the Method Editor and select Override Variable
- The variable in the Subclass is now a different variable to one in Super Class
- This can be done with Class and Instance variables
- Beware copy and pasted fields and code lines can **automatically** override variables
  - To avoid this comment code before copying to clipboard, paste in and then uncomment (don't forget to uncomment in source method)

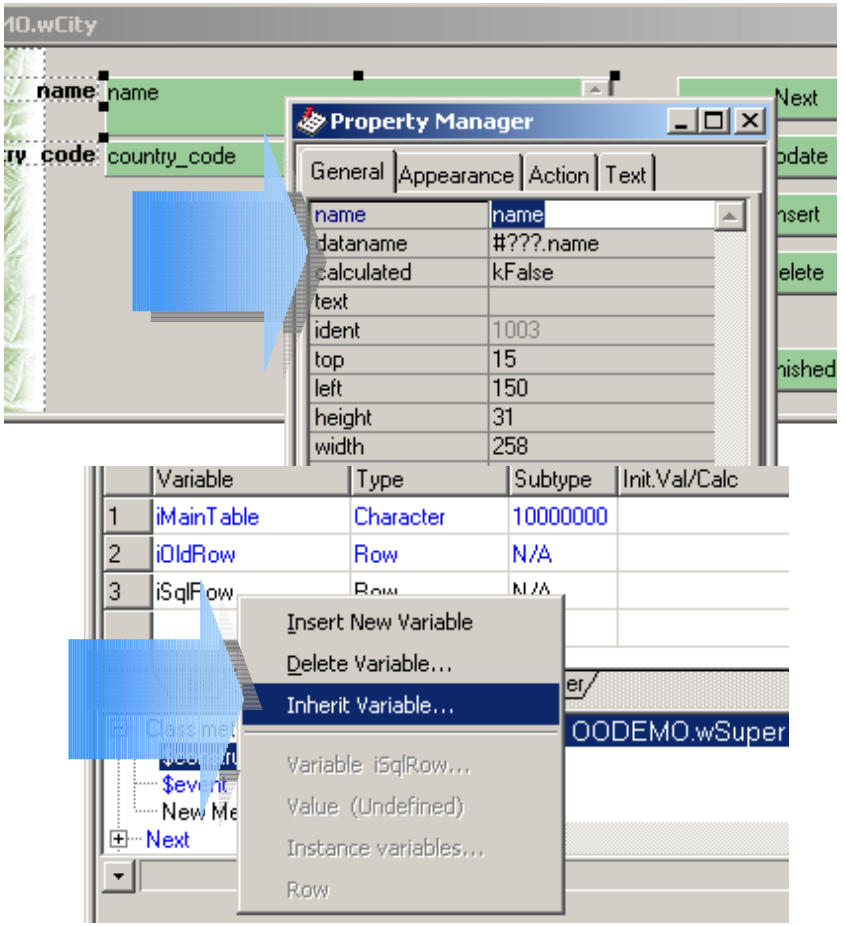


Accessing Overridden Variables

- Overridden variables are still in scope
- To access the Super Class variable from a Subclass method use `$inherited.myVar`
  - Note that this will always access the variable from the next level up in the inheritance tree
- To access the Subclass variable from a Super Class method use `$cinst.myVar`
  - Note that this will always access the variable from the lowest level in the inheritance tree

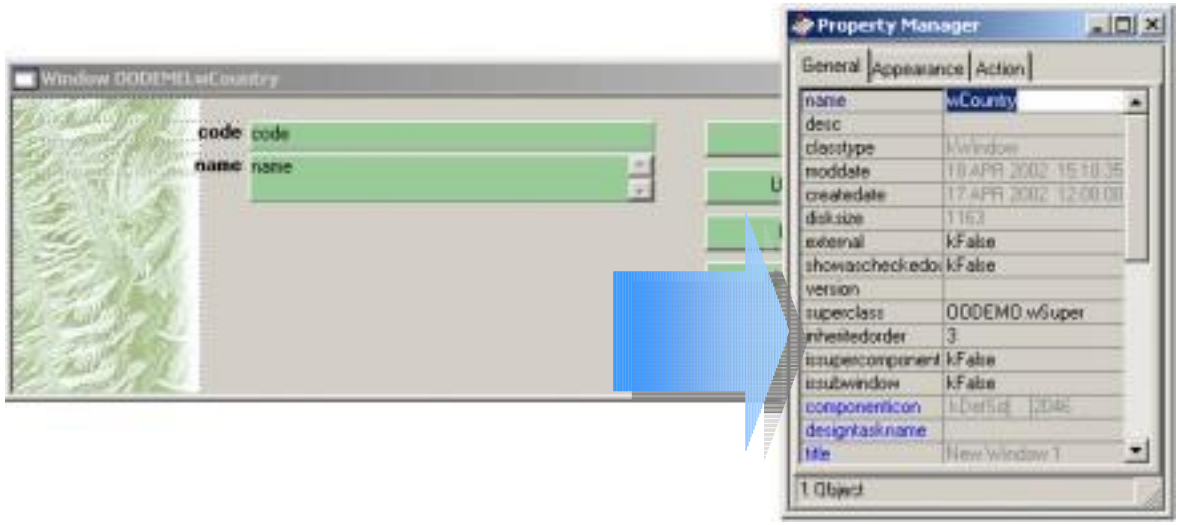
Inheriting Variables

- A variable that has been overridden in a Subclass can be inherited
- Right-click on variable in the Method Editor and select Inherit Variable
- To do this the Subclass variable is removed and references to variable are affected (changed to #???)



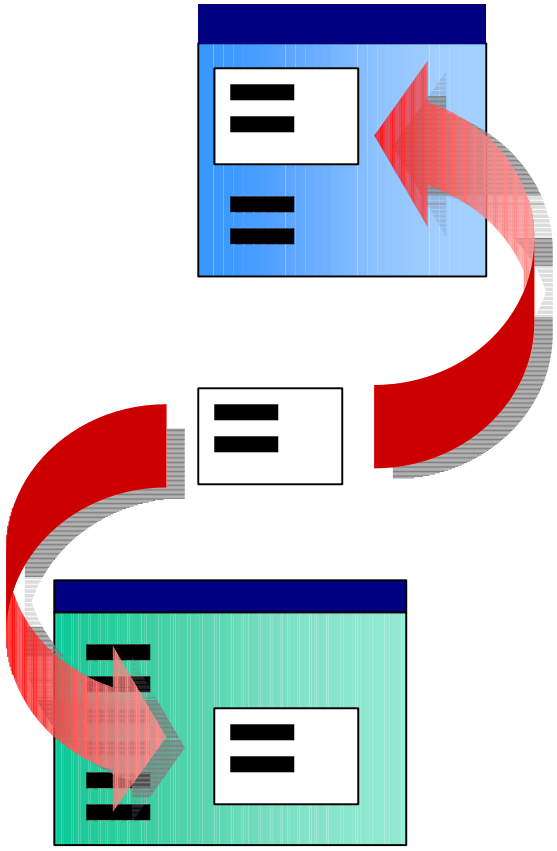
Inherited Order

- Inherited fields appear first in tab order
- The tab order of the first inherited field can be set using the $inheritedorder property
- This needs to be set for each level in the inheritance hierarchy



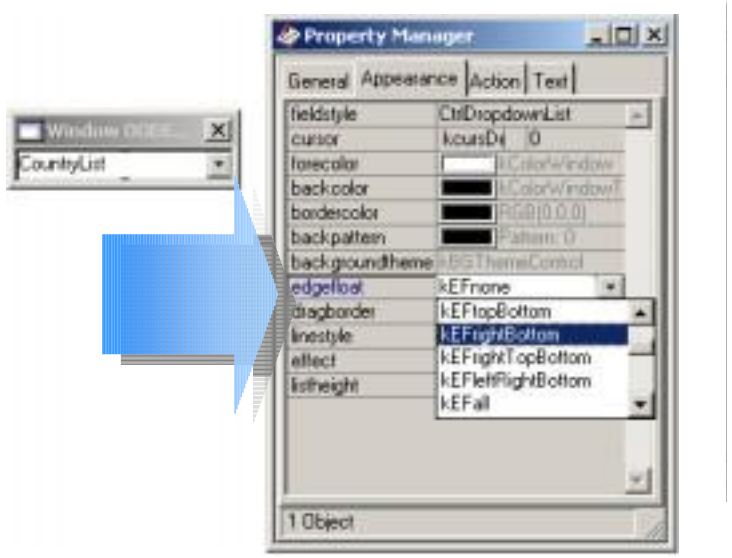
Subwindows

- A Subwindow is an aggregation
- It is a window class and can contain normal field and background objects
- It may contain many fields or just one or two
- A single Subwindow can be reused on any number of other window classes
- It is an alternative reuse mechanism to Inheritance
- The Omnis Studio “version” of an ActiveX control



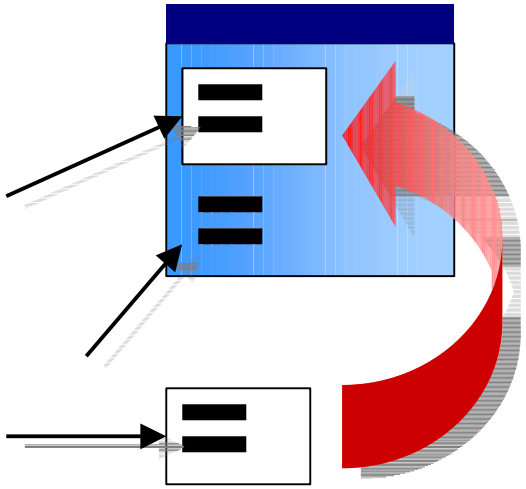
Creating a Subwindow

- Create a window as normal
- Size the field so that it fills the window
- Set the $issubwindow property to kTrue
- Hint: for subwindows that implement metafields use floating properties on the objects inside it to ensure the fields resize automatically



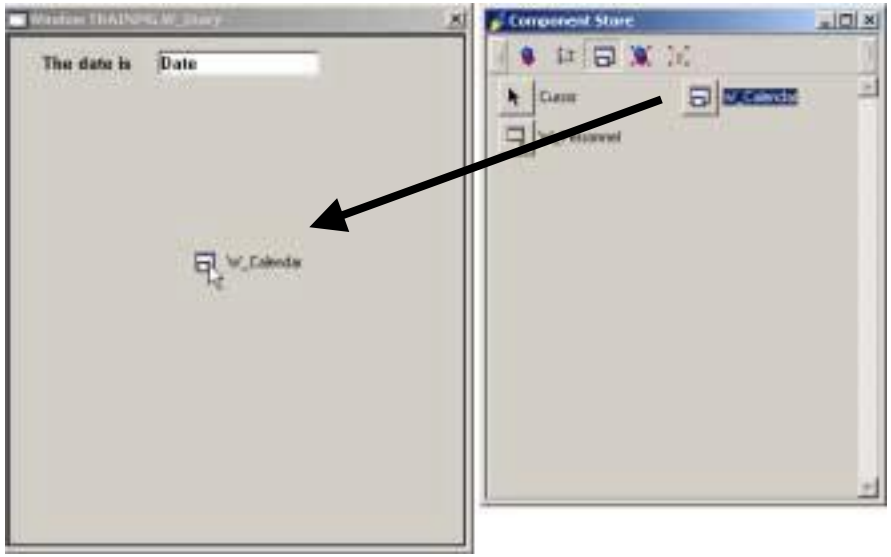
Subwindow Interface

- A Subwindow needs to be able to communicate with its parent window and vice versa
- We need to build an interface of public methods
- Inside a subwindow field method on parent window
  - $cfield refers to the subwindow instance
- Inside a subwindow instance method
  - $cwind refers to the parent window instance
  - $cinst refers to the subwindow instance



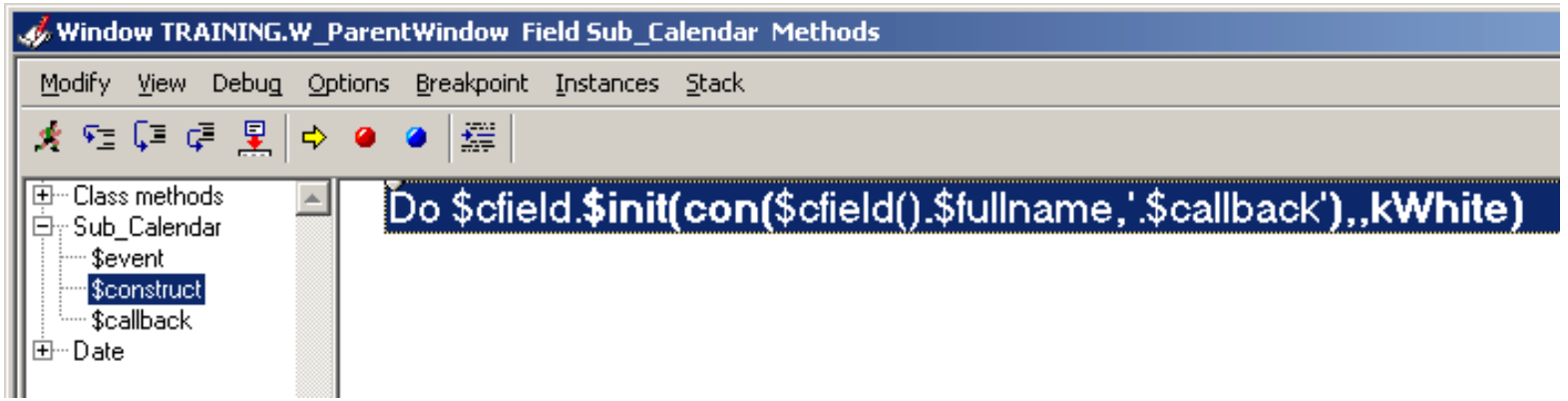
Using a Subwindow

- Select the Subwindows icon on the Component Store toolbar
- Drag required subwindow across and drop it on the window
- Set up any required Interface calls from the parent window to the subwindow



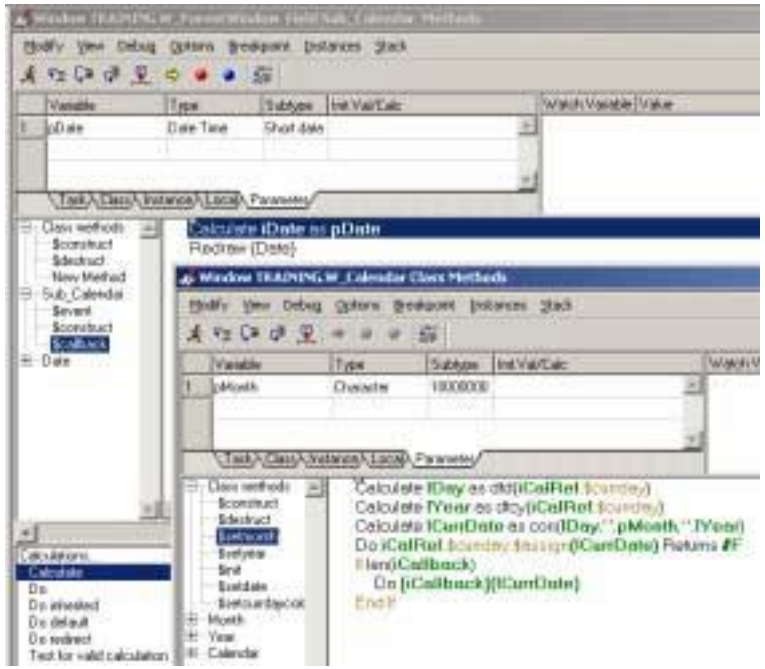
Initializing a Subwindow

- Initialise a subwindow by calling a public method from the container field $construct method on the parent window
- Pass initial data as parameters
- This data may be used to initialise data to be displayed inside the subwindow and other visual aspects of the subwindow instance (e.g. colour of fields, enable/disable etc.)



Subwindow Callback Method

- When an event occurs inside a subwindow instance the event is not accessible to the parent window event handlers
- We need to implement a callback to the parent window
- When subwindow init is called, pass the full notational path of a Callback method that should be called when an event occurs in the subwindow
- Path to callback method is cached by $construct method in an instance variable
- Callback method may receive data from subwindow as parameters



Combining Subwindows

- Subwindows can be combined with other fields on the same window
- Other objects communicate with the Subwindow via the public interface
- The Subwindow can communicate with other objects via the callback

Table Classes

- Instantiated as a row or list variable
  - Table instance and row or list variable are one and the same
- Instance created when row or list is defined

```
Do myRow.$definefromsqlclass('myTab',,p1,p2)
```

  - Define from table class, not schema or query
  - Parameters may be passed to $construct method, note ',,'
- Incorporate a data interface via row or list
- Methods provide an abstraction of SQL
- Enable you to separate the data logic from the GUI
- Public methods of table available via list or row

```
Do myRow.$select(con("where myCol = '",myRow.myCol,"'"))
```

- In table method $cinst refers to instance and row or list variable
- Instance destroyed when variable is cleared
  - $destruct method does not run automatically

Object Classes

- Similar capabilities to Table class instances but no data interface (row variable or list)
- Instantiated as a variable of data type object
- May be statically or dynamically defined
- Instance destroyed when variable is cleared
  - – $destruct method does not run automatically

Object Instances

- Static definition
  - Define a variable of required scope
  - Set data type to Object and set subtype to your object class
  - $construct method runs the first time that you call a method
- Dynamic definition
  - Define a variable of required scope
  - Set data type to object, leave subtype empty

```
Do $clib.$objects.myObj.$new() Returns myVar
```

  - $new method performs the same function as $open
  - $construct method runs when the instance is created by $new

Omnis

Storing Objects in the Database

- Since object is a standard data type object instances may be stored in a database column
- Cannot identify object instances containing specific values without reading all instances
- By default
  - The objects data (instance variables) are stored in the database
  - The methods are provided by the object class and may change over time
- To save the methods in the database
  - Set the object class $selfcontained property to kTrue
  - Takes a “snapshot” of methods in the class
  - Stored objects occupy more space

Omnis

Summary

- Application based on software objects
- It is a different way to construct systems
- It is a paradigm shift
- It is a technology
- More importantly it is a methodology
- Benefits
  - Reusability
  - Consistency
  - Maintainability
  - RAD (eventually once super classes have been created)