DATALINE SOFTWARE

i5 Developer's Guide

Suite 6 • Clarence House • 30-31 North Street • Brighton • BN1 1EB UK

Phone 01273 324939

enquiries@dataline.co.uk • www.dataline.co.uk

Trademarks

i5 is a registered trademark of Dataline Software Limited (Dataline).

Adobe is a trademark of Adobe Systems Incorporated.

Apache Web Server is a trademark of Apache Group. This product includes software developed by the Apache Group for use in the Apache HTTP server project, http://www.apache.org.

"IT IS EXPRESSLY AGREED THAT DATALINE SHALL NOT BE IN ANY WAY RESPONSIBLE FOR THE COMMERCIAL SUCCESS OF THE APACHE WEB SERVER PRODUCT. DATALINE OFFERS THE APACHE WEB SERVER PRODUCT "AS IS" WITHOUT WARRANTIES OF ANY KIND, AND DATALINE SHALL HAVE NO LIABILITY OR RESPONSIBILITY FOR DAMAGES OR LOSS RESULTING FROM OR IN ANY WAY CONNECTED WITH THE APACHE WEB SERVER PRODUCT."

Java is a trademark of Sun Microsystems Inc.

Microsoft, Internet Explorer, Internet Information Server, Visual Basic , Visual Studio, Visual Studio .NET, Win32, Windows, Windows NT, Windows XP, Windows Vista are registered trademarks or trademarks of Microsoft Corporation in the USA and/or other countries.

All other products or service names mentioned herein are trademarks or registered trademarks of their respective owners.

Copyright

Copyright © 2009 by Dataline Software Limited. All rights reserved. July 2009

Contents

Contents 3
1. Getting started with business logic 4
COM and .NET 4
Components of an i5 COM business logic application 5
COM classes and the i5 Business Logic Server (BLS) 5
COM libraries 5
i5 COM & .NET programming interface (API) 6
The i5 Business Logic Administrator (BLA) 6
Working with .NET 6
Setting up a .Net Project 7
Writing Some Code 8
Compiling and Uploading Business Logic 10
Debugging 11
Summary 14
2. Using events to build dynamic pages 15
i5 Events 15
Creating and modifying i5 field objects 16
Handling user input 21
Using the OnRequest event 25
3. Managing User State using Container Objects 27
User State Management 27
State variables 27
Summary 33
4. Databases, connectivity and child containers 34
Database Connectivity 34
i5 Child table events 40
Writing business logic code for i5 child container events 41
5. Building plug-in components 46
A new plug-in component 46
Configuring components with properties 48
Creating a plug-in email form 48
About plug-in component class properties 51
6. JavaScript Issues 55
Using JavaScript in i5 to do data validation 56
Exploiting Properties, Methods and Event Handlers 57
Concluding Remarks 62

1. Getting started with business logic

The real value of i5 as a development environment lies in the way that it allows the programmer to extend the point and click capacities of i5 Studio with bespoke business logic code. The i5 3.5 API provides the developer with a complete set of classes for processing data, generating pages dynamically and generally providing the control required to organise sophisticated web applications.

Business logic code can be written in a multitude of languages. The i5 3.5 .Net API is written as a set of .Net wrappers around COM code and as such can be used with any native .Net programming language. This is unlike i5 version 3.0, which required the generation of complex interoperable code each time you developed a project.

With version 3.5 the decision has been made to develop all the code examples given in this chapter – as well as elsewhere in this Guide – in C#. However, it is not difficult to translate code written in C# into other languages, whether these are .Net compliant or not. Additionally, the *i5 3.5 .Net Library Reference*, which accompanies this Guide, not only provides helpful documentation on the objects and methods available to the programmer for development purposes: it also incorporates code fragments written in Visual Basic as well as C#, thus providing a basic knitting pattern for translating between these two popular programming languages.

The basic aim of this chapter is to introduce you to the key processes involved in writing business logic for i5 applications. We will do this by first, exploring some key features of the underlying technology used by i5 and second, by looking at what you need to do to get i5 to say 'Hello, World'.

COM and .NET

i5 and its forerunner net.db were both built to allow the developer to work with COM objects when writing business logic. However, unlike net.db, i5 version 3.5 is designed to be fully interoperable between COM and .NET. Here we explore a number of aspects of the COM technology and its interaction with i5 – by itself and within the context of the .NET framework. You can skip the following discussion if you want to get straight in to the programming – it's not essential. However, for a fuller understanding of what goes on 'under the bonnet' in the API, the discussion helps.

COM, if you didn't know already, stands for Component Object Model and was created in 1991 as a proprietary document integration and management system for the Microsoft Office suite. It has since become a framework for creating and using components, making software easier to write and reuse, and providing a wide choice of services, tools, languages and applications. There are a number of useful introductions to working with COM, if you are not familiar with it, one of the most thorough guides being provided by Don Box in his *Essential COM*.

There are a number of features of COM which make it a sensible choice for developing web applications. In addition to allowing the developer to work with a wide range of programming languages, it has a number of other advantages:

- Being integrated with a number of development tools;
- Permitting a standardized use of services, which are the same regardless of location;
- Providing flexible security;

- A mature specification and reference implementation;
- In an increasingly .NET-oriented world, retaining COM as the basis of an interoperable API makes it easier to achieve the kind of backward compatibility needed to work with legacy systems.

Of course, the really important thing about COM – like .NET but for rather different reasons - is that it promotes code re-use and platform independence. It is thus a crucial factor in the development of the kind of multi-tier web applications which i5 was designed to build.

Components of an i5 COM business logic application

There are a number of components which go to making up an i5 business logic application.

i5 COM Business Logic Server (BLS). As its name suggests, it is the BLS which is responsible for dealing with requests from the i5 Object Engine, doing all the processing of information specified by the developer in his or her business logic code and sending the results of this processing back to i5;

i5 Object Engine. The Object Engine issues requests to the BLS and to the database server, integrates the results of these requests and turns them into html ready for rendering by a browser;

i5 3.5 .Net API. The API contains all the classes used by i5 to create the objects which will be rendered on i5 pages as well facilitating interactions with a database and so on;

COM Classes. The developer-specified code determining particular forms of data processing, page rendering and so on;

Business Logic Administrator (BLA). The BLA is a tool used by the developer and/or by a system administrator for uploading and downloading code used in i5 applications, setting up debugging processes and associated logistical issues.

COM classes and the i5 Business Logic Server (BLS)

A typical business logic application is distributed over many computers and therefore requires a method for the remote execution of business logic on each computer. The i5 Business Logic Server (BLS) is the tool that i5 uses to achieve this. Every computer that is used to process business logic in an i5 application has an i5 COM BLS resident in memory. Each COM BLS executes business logic code and sends the results back to i5.

There is, theoretically, no limit to the number of computers which can be used to process business logic. As a consequence, a COM business logic application is highly scalable and it is an easy matter to distribute processing over a number of machines as the popularity of a website increases. For the same reason it is also possible to write business logic which consumes high levels of resources and processing power without this having an impact on other business logic processes.

COM libraries

For a COM BLS to access COM Classes, the COM Classes must be stored in COM Libraries, which take on the form of Dynamic Link Library (DLL) files. These can be created using Visual Basic, C++, CTD, Delphi, or other development environments.

i5 COM & .NET programming interface (API)

The i5Net35 API is, as we have already stated, the collection of classes that are used to program business logic. You can find a detailed account of all these classes in the *i5 3.5 .Net Library Reference* guide, so we won't go into a lengthy discussion about them here. However, you will encounter a number of these classes in this Developer's Guide, so it is probably worth mentioning them all here, with a very brief description.

Class	Description
Container	Contains i5 page objects (buttons and fields, images, text, other containers and so on). Can take the form of pages, of tables on pages, as well as 'groups'.
Field, CheckBox, Image, ListBox, Text, PageLink, & RadioGroup	Produce instances of a range of familiar html-based objects and offer a range of methods for manipulating these objects
Session	Provides access to user information and allows for precise control over aspects of the user's interactions with i5 pages.
Database, SQL, HTTPForm & Mail	Enrich the programming environment by allowing the creation of things like handles to databases, HTTP form methods etc.
Dimension & Coordinate	These provide precise control over the layout of objects on an i5 page

The i5 Business Logic Administrator (BLA)

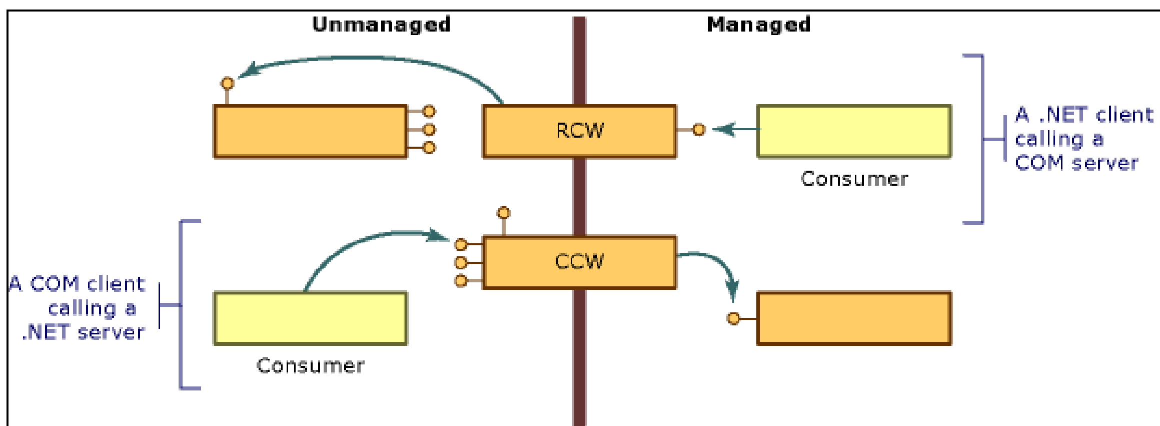
The Business Logic Administrator (BLA) is a browser-based tool that that allows you to see Business Logic Servers connected to i5, and the business logic code classes/methods which are available to i5 from each BLS. It is also used to install new Business Logic Servers and new or updated COM libraries to those servers.

The BLA – which can be opened by clicking the icon on the i5 Studio toolbar or from the Windows Start menu - allows COM libraries to be installed remotely over any TCP/IP based network. This means that business logic code can be written and tested on a single development platform and then deployed to remote BLSs from a single location.

Working with .NET

Having pointed out that i5 works with COM objects, the question arises as to how business logic code written in .NET languages such as C# can interact with COM classes. The simple answer is *interoperability*. Recognising the desirability of (and need for) backwards compatibility (would you really want to completely re-write your large and sophisticated library of COM code using .NET? aren't there more pressing calls on your time?), the .NET framework allows for communication between COM and .NET. It does this by the provision of code *wrappers*. Code wrappers are needed in order to negotiate the difference between the ways in which COM and .NET manage the lifetime of the objects that they produce. They are also needed to do a few other things but the detail is not relevant to what we are doing here. To enable communication between COM and .NET, .NET provides two kinds of wrapper: *COM Callable Wrappers* (CCW) and *Runtime Callable Wrappers* (RCW). The former allows a COM client to call into the managed

world of .NET code. The latter, as you might expect, allows a .NET client to call into the unmanaged world of COM code. The following diagram summarises this situation:



So, what does this mean in practice? Does it mean you have to write lots more code? Well no, not really. In the last version of i5, version 3.0 you had to be very careful in how you set projects up and had to write your own interface code to facilitate interactions between COM and .NET. In version 3.5, by contrast, the work of generating the necessary extra code is taken care of in the API, freeing you up to concentrate on writing the code needed directly to implement your business logic functionality.

Let's look then at setting up a .NET project to produce interoperable business logic code that will say 'Hello, World'.

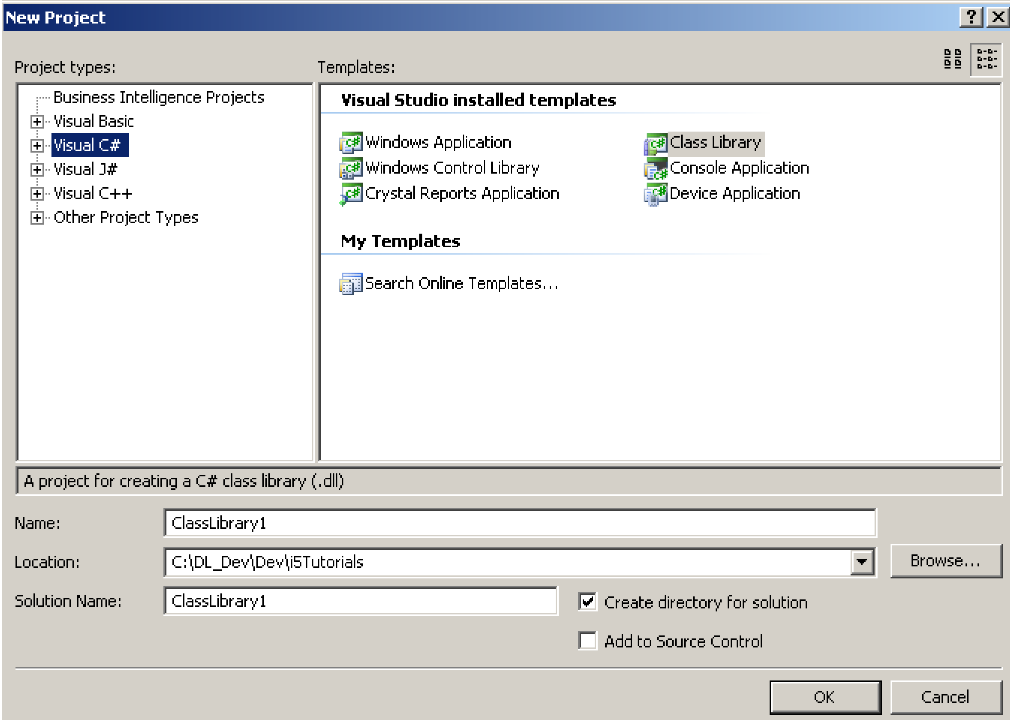
Setting up a .Net Project

As we have already suggested, writing business logic for i5 using the .Net framework is very much a matter of being sure that you have set up your project in Visual Studio .Net correctly. In the first instance it is necessary to generate a Runtime Callable Wrapper so that the .Net Common Language Runtime can interact with the i5Net35 API (see the diagram above). We will now walk through the steps required to do this.

Please note that all of the code examples in this Guide are written using Visual Studio .NET 2005. The first thing that you need to do is to open up Visual Studio and click on the Create Project option. This will open up a Project dialog box that will allow you to set key project options.

We are going to be developing this project in C# so we will select the C# Class Library Template from the New Project dialog box. This will ensure that we generate class library files and not an .exe or a set of forms.

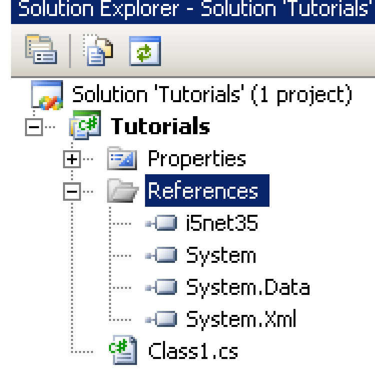
We suggest that you call your project something reasonably sensible and meaningful – Tutorials, for example. When you change the name of the project from the default, note that the solution name will be updated as well. It is also a good idea to keep all your development work in a meaningful directory structure.



In the example above, we are locating our tutorials work in a directory called Dev\i5Tutorials\. Ignore the 'Add to Source Control' option unless you have source control software on your machine.

Once you've created a new project, Visual Studio will give you a skeleton of code as a starting point for your business logic. Before you can start working with this though, you need to add a reference to the i5 class library from the 'Solution Explorer' of your project. This will be located on the right hand side of the screen.

You can add a reference by right-clicking on the References node in the Solution Explorer, selecting 'Add Reference' and then navigating to the i5 class library:



You can verify that a reference to the i5 class library has been added to your project by checking in the Solution Explorer window, as per the screenshot to the right here.

We are now almost ready to write some code. However, before we do so, we need to set up a page in i5 to use with our business logic. So, after logging in to the i5demo database from i5 Studio, create a new book (call it something like 'Business Logic'), add a section called 'TutorialOne' (without specifying a datasource) and then within that section, a Detail page called 'pgHello'.

Having created a new page in the Designer, we can return to our C# Tutorials project. At this point you should change the name of the default C# class that was set up when you first created the project to pgHello.cs. You can do this from the Properties window of the .Net environment.

Writing Some Code

OK, now we have set up our project in Visual Studio, we are ready to write some code. The following code is sufficient to demonstrate some basic rules about writing code for i5 applications. Add these lines of code to the pgHello.cs file in Visual Studio.

```
using System;
using Dataline.i5;

namespace Tutorials
{
        public class pgHello: Events
        {
                protected override void OnCreate(Container page)
                {
                        base.OnCreate(page);
                        using (Text myText = page.Text("txtHello"))
                        {
                                myText.StrValue = "Hello World";
                                myText.TextSize = 2;
                        }
                }
        }
}
```

There are a number of things to observe here. The first two lines tell the compiler not only that we wish to make use of the C# System library but also that we wish to use the Dataline i5 API. The compiler will throw an error if this library hasn't been added to the list of references for this project. The next line meets the C# requirement that namespaces be used. Namespaces help organise your code and prevent possible ambiguities arising. You can also refer to a namespace with the 'using' keyword. The fourth line of code public class pgHello : Events gives a name to the class that we are creating, corresponding to the name of a page in our i5 book. It also tells the compiler that the class inherits from the i5 Events base class. This makes a series of methods – such as the OnCreate method – available to us. The keywords 'protected', 'override', and 'void' in the subsequent line do a number of things. Our method is not going to return anything, so we give it the return type 'void'. The 'override' keyword allows us to modify the abstract OnCreate method defined in the Events class. The 'protected' keyword means that the method that we are defining here is only accessible within the pgHello class or any subsequent class which might inherit from it. Within the body of the method we are basically doing two things. First, we invoke the OnCreate method of the Events base class. Then we create an i5 Text object and set some of its properties. The 'using' directive here allows us refer to these properties without having to fully qualify the identity of the object to which these properties belong in each line of code. Basically, this code tells i5 what to do when creating pgHello in the browser: put the piece of text "Hello World" on a webpage.

The previous version of i5, version 3.0, required a slightly more complex programming construct to do this – as mentioned previously, you were expected to write interfaces for all the classes that you created, you had to write in a GUID for each class to ensure that it was unique, and you had to set your project options very carefully, being sure that the code assemblies you were producing were 'COM visible'. Because the new i5 API is compiled as a set of .Net .dll files there is no longer any need to take these extra steps.

If you are unfamiliar with Visual Studio, you should note that it uses colour coding to distinguish between different elements in a programme. Dark blue lettering indicates keywords in the programming language; light blue indicates classes in the i5 API; green indicates programmer

comments; brown is used for string data that will be either rendered as such or used to name page components for i5; everything else in the code that you write is rendered in black.

Compiling and Uploading Business Logic

In version 3.0 of i5, uploading business logic written in a .Net language was always a little fiddly as you had to upload more than one file onto the Business Logic Server, something that wasn’t possible with the Business Logic Administrator. In version 3.5, uploading .Net business logic is exactly the same as it is for COM-based languages such as Visual Basic: you just upload the file containing the business logic that you have written. Before doing that, you do of course have to compile the code that you have written – you can do this by clicking the 'Build Tutorials' item from the Build menu on the main Visual Studio toolbar. Assuming your code compiles successfully, we can now get the Business Logic Server up and running and use the Administrator to upload the code.

Click on the cogwheel icon from the range displayed across the top of the left hand frame of the Designer. If this icon is dimmed, the Business Logic Server is not running. You will therefore need to start it, something that can be done in one of two ways. If you are running the BLS in console mode, from the Windows Start Menu, go to Programs>i5>Server>i5 Business Logic Server. If you are running the BLS in service mode, go to Settings>Control Panel>Administrative Tools>Services.

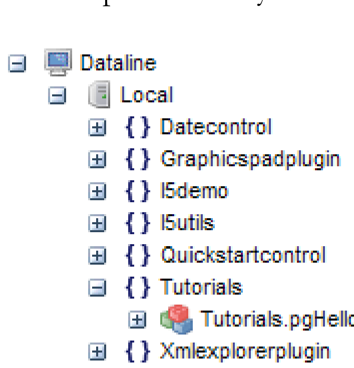
With the BLS running, you can now open the BLA by clicking on the cogwheel icon (or from the Programs menu by navigating to i5>i5 Business Logic Administrator). On first opening the BLA you will be required to log in, using a Windows NT domain user name and password that has membership of the i5 Admin User group. If you are not sure whether you have i5 Administrative privileges, you can check membership of the i5 Admin user group in the Users and Passwords directory in the Windows Control Panel.

Dataline
Local
{ } Datecontrol
{ } Graphicspadplugin
{ } I5demo
{ } I5utils
{ } Quickstartcontrol
{ } Xmlexplorerplugin
New/Update Library
New Server

Once in the BLA, an Explorer view is displayed showing the Business Logic Servers being used by i5. In the example here, only the “Local” Business Logic Server is in use.

Clicking on the New/Update Library link will display the New Library page in the right hand frame of the Administrator

The Tutorials file needs to be uploaded, so click Browse to locate it (unless you've changed the Project settings in Visual Studio, it will be in the Release/obj subdirectory of the directory that Tutorials was saved in) highlight Tutorials.DLL and click Open. Click the Upload Library button - this sends the DLL to the BLS and installs it.



Once installed, Tutorials will appear in the Explorer view under the Local Server, and when you expand it, its classes and methods will be displayed –as you can see here.

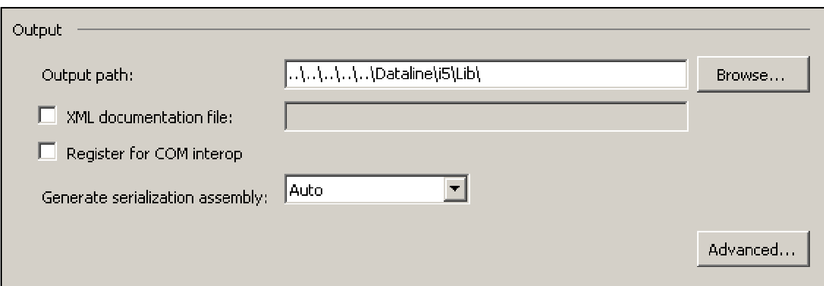
With the Tutorials .dll now loaded safely into the BLS, you can reference it on your i5 pages, thus instructing i5 to trigger some logic every time the specific page event for the named page occurs. To do this you should: return to the Designer and click on the Logic property tab of the page pgHello, click Next, and select the Tutorials from the dropdown list, click Next again, and select the pgHello from the dropdown list of classes, then save your changes.

You should find that the text 'Hello World' should now appear on the page in the right hand frame of the Designer.

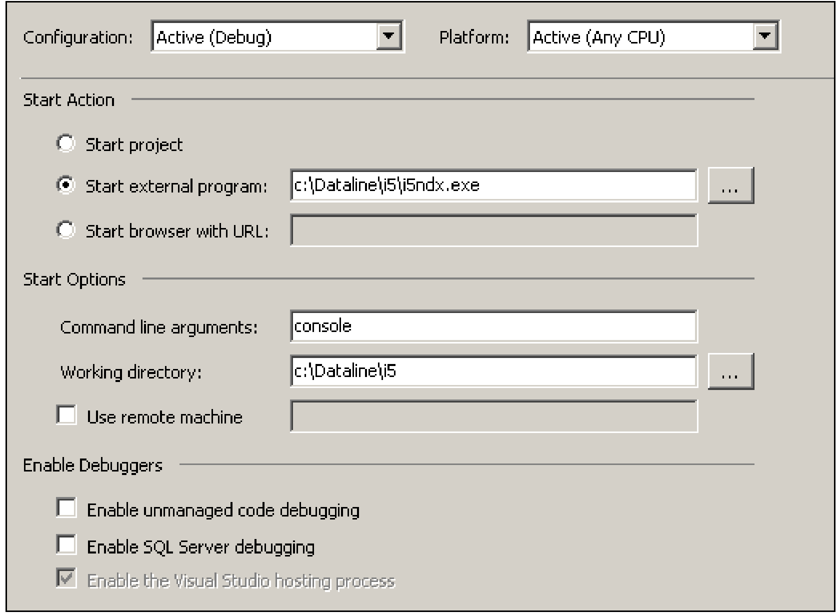
Debugging

The code example that we have looked at here is really very, very simple, so it would be somewhat surprising to discover that there are problems with it. However, it is unlikely that you will always be writing code this simple (and in any case, you don't need to use business logic to display the message 'Hello World' on an i5 page). It might be helpful at this point to look at how you can set Visual Studio up so that you can use its excellent code debugging facilities with i5. To do this, you will need to edit the properties of the Tutorials project. Click on the Project menu from the main Visual Studio menu and navigate down to the last menu item 'Tutorials Properties'. This will open up the Properties settings for the Tutorials projects. This multi-tabbed box allows you to define a number of key settings for your project.

In the first instance, we need to tell Visual Studio where to put the business logic code that it compiles for you. To simplify matters, let's tell Visual Studio to put the compiled code in the Lib directory where i5 normally stores business logic files. To do this, go to the Build tab of the Tutorials Properties box and set the output path to point to the Dataline\i5\lib directory on your machine:



Having set the output path of your project, you can now set its debug properties, so navigate to the Debug tab of the Tutorials Properties box and change the settings so that they read as follows:



This tells Visual Studio what to do when you wish to debug code – specifying which program it should use to run your code. You shouldn't need to make any other changes (although if you are working in a multiple user/multiple machine setup, there are some complications that need to be addressed – we will save that for a later chapter).

You should now be ready to use the debugger. Go back to the pgHello.cs screen in Visual Studio. We want to set a breakpoint in the code – do this by clicking in the left hand margin of the screen, level with the line using (Text myText = page.Text("txtHello")). A breakpoint should now appear on the screen:

```
using System;
using Dataline.i5;

namespace Tutorials1
{
    public class pgHello: Events
    {
        protected override void OnCreate(Container page)
        {
            base.OnCreate(page);
            using (Text myText = page.Text("txtHello"))
            {
                myText.StrValue = "Hello World";
                myText.TextSize = 4;
            }
        }
    }
}
```

Now, if you go to the Debug menu and select the Start Debugging option, you should find, first, that the i5.exe program starts (in a console window), then, if you preview the i5 page in i5, execution of the program will halt at the breakpoint you have set in your code (which will now be highlighted in yellow):

```
using System;
using Dataline.i5;

namespace Tutorials1
{
    public class pgHello: Events
    {
        protected override void OnCreate(Container page)
        {
            base.OnCreate(page);
            using (Text myText = page.Text("txtHello"))
            {
                myText.StrValue = "Hello World";
                myText.TextSize = 4;
            }
        }
    }
}
```

If you then click on the 'Continue' option (the green arrow on the main toolbar in Visual Studio) your code will then continue to execute and the i5 page will display with the text settings as you have defined them in the code. Visual Studio provides a 'Pause' function that allows you to pause the execution of your code so that you can make changes to it on the fly, as well as a range of other options to allow you to inspect variables in your code and so on. We can't examine the facilities that the debugger offers here – you will need to refer to Microsoft's own documentation to learn more about how to use this aspect of Visual Studio. All we wanted to do here was show you how to set up your project so that you *can* debug it.

In addition to the debugging facilities provided in Visual Studio, i5 does offer some further support for finding out what happened when something goes wrong. The main \Dataline\i5 directory contains a log file – i5ndx.log - which records Business Logic Server errors. A typical message in this log file might read "10/01/09 11 56:11 Failed to load 'I5Utils'. No such interface supported". Admittedly not the most detailed of messages but a helpful pointer all the same. By default, the log file is located in the main i5 directory. However, you can configure your i5 installation to use a different directory for error logging. The i5 configuration file – i5ndx.ini (also located in \Dataline\i5) contains a 'LogDir' property which by default is omitted from the configuration. The following settings for i5ndx.ini will write all Business Logic Server errors to the directory C:\MyBusinessLogicErrors\:

```
Port=2021
InstallLibDir=InstallLib
LibDir=Lib
LogDir=C:\MyBusinessLogicErrors\
ServerStopTimeout=20
maxClientQueue=50
maxClientThreads=20
```

It is unlikely that you will need to use the debugger to debug the code examples that we develop in this guide, so we suggest that when developing the code samples, you always compile using the 'Build' option rather than the 'Start Debugging' option in Visual Studio. However, should you need to, you will at least know how to run the debugger now.

Summary

If everything has worked up to this point then you have grasped the basic procedures involved in writing and installing business logic for use in i5 applications. As you can see, it really isn't that difficult - in keeping with the general i5 philosophy of rapid development. Of course, with complicated business logic there are plenty of things which can go wrong but the basic procedures are fundamentally the same as they are here. Having said that, we can now start to look at some of the things that we can do by marrying i5 and business logic.

2. Using events to build dynamic pages

In the previous chapter we discussed the bare essentials of what you need to know and do in order start writing business logic. In this chapter we do two things. First, we discuss the concept of a *page event*, which is fundamental to working with the i5 API effectively. Secondly, we use this discussion to introduce some of the key functionality that you will want to program into an i5 application.

i5 Events

In order to function effectively, i5 needs to know under which circumstances business logic code should be executed. These circumstances can be characterised as belonging to one of a number of categories of events. Events are triggered as something occurs in an i5 application. For example, navigating to a particular page in an application will trigger a page create event, clicking on a page link will trigger a page submit event and so on. Business logic code is linked to programmer-specified page events and the code that is executed is dependent on the event that has taken place: in the previous chapter we utilised the page create event in order to write 'Hello, World' to the screen in a book.

Clearly a good understanding of what different events allow you to do is important to writing efficient and effective business logic, so in this chapter we will explore three of the main events available when programming pages - create, submit and request.

> *Event handling code can be written for i5 pages, tables, groups and plugin components. Technically speaking, these are all in fact containers for other components. So any container can have business logic events assigned to it. The concept of a container is discussed in more detail in the next chapter.*

Page OnCreate event	Every time a user goes to an i5 page or an i5 page is refreshed, the OnCreate event is triggered. This event takes place immediately prior to rendering of the page.
Page OnSubmit event	This event is triggered when a user clicks on an i5 Page Link button and all of the values entered by the user on the page are sent to i5 for processing.
Page OnRequest event	Each time a user goes to an i5 page or when an i5 page event is refreshed, this event is triggered, taking place immediately prior to the OnCreate event.

Calling order of page events

When a user clicks on a Page Link in their internet browser, i5 page events are called in the following order:

OnSubmit	Data from the page is submitted to i5.
OnRequest	i5 requests access to the page that the Page Link is targeting.
OnCreate	i5 creates new target page.

> *When a user comes to a page in an i5 website from outside of that website – via a URL to its home page, for example - the Submit event is not triggered, but the Request and Create events are.*

Event handlers

Writing business logic code in i5 is effectively a matter of writing event handlers, in a manner analogous to the event handlers that one might write in JavaScript, for example. When writing in C#, the methods that you write will be class methods – i.e. you will create classes in C# and then write methods for those classes. It is these class methods that the i5 business logic server will invoke.

In order for i5 to invoke the correct methods for a page event, each page should in the first instance be assigned its own business logic *event handling class*. The name of such a class should be declared as the name of the page for which it is written. The event handling code is then written as a number of *methods* of that class. We saw that in the very simple 'Hello, world' example in the previous chapter, which was written to trigger on the page create event.

Here's another example. Given an i5 page with the name pgSearchResults, a class pgSearchResults would be created and the event handling code for its page submit event would be declared as:

```
public class pgSearchResults : Events
{
        protected override void OnSubmit(Container page)
        {

                base.OnSubmit( page )
                ...event handling code for submit event goes here
        }
}
```

Notice how the method is passed an i5 Container class as a parameter. The Container class represents the contents of the i5 page and is used to access and to modify the properties of the objects that it contains. It can also be used to create new objects on a page. We will look at these possibilities in more detail later in this chapter.

Alternative Notation

Previous versions of i5 allowed you to use an alternative notation for the specification of page events: the API allowed you to add the name of the page followed by an underscore as a prefix in order to identify which event was to be fired for which page in your code. For example, the OnCreate event of a page called pgTest could be written pgTest_Create(Container as Parameter). It is recommended that you do <u>not</u> use this notation with i5 version 3.5.

Creating and modifying i5 field objects

The design of the i5 API is such that the properties of any object on an i5 page can be fully accessed and - if required - modified in business logic by passing a reference to an i5 page (or other 'container' object) into to a page (or container) event method and then referring to the object in question on the page. As a result it is possible not only to create objects on a page and modify their properties using business logic but also to modify the properties of objects that have been added to a page previously using the Page Designer.

Fields on a page that are linked to database columns - i.e. they have their *Column* property set - are populated with data from the database just before the OnCreate event and objects on the page will not be drawn until the OnCreate event has completed.

Accessing the properties of an existing object

Accessing the properties of an object that already exists on an i5 page requires getting a reference to the object. To get a reference to an object, a method of its container must be called and this method will return a reference to the desired object. The container method to call depends on the type of the object to be accessed, so the name of the method represents the object type. To access a field object, for example, the 'Field' method of the container would be called, as in the following code fragment

```
protected override void OnCreate(Container page)
{
        base.OnCreate(page);
        using (Field myField = page.Field("dfUserName"))
        { // gets reference to 'dfUserName' field from container
        ...
```

Each i5 object has properties that can be retrieved and changed. Extending the above code will allow the Title property of the field to be stored and retrieved, as in the fragment below:

```
        string sTitle;
        .Title = "Enter Your Name";  // set the title
        sTitle = myField.Title;  //get the title back
        //sTitle is now "Enter Your Name"
        }
}
```

Some properties are read-only – the Name property is an example of this. See the Function Reference section for further information

Tutorial: Modifying objects using business logic

A simple tutorial should be enough to demonstrate how the properties of an object in a container can be altered using business logic. Here we show how to alter the properties of an editable data field.

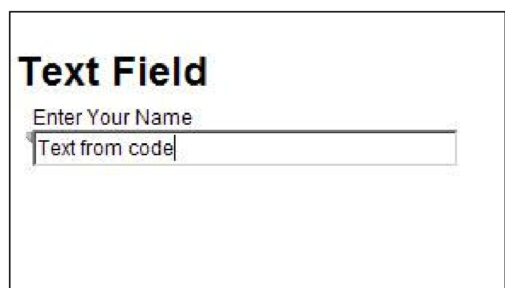
Create a new section in your i5 Tutorials book named TutorialTwo and within that section, a new detail page named pgTextField, with the title Text Field**.** Then add a data field called dfField with the title Text, so that you have a page that looks something like this:

In Visual Studio create a new class module called pgTextField. You can do this by right-clicking on the Tutorials project icon in the Solution Explorer, navigating to Add > New Item and selecting the C# Class component from the Add New Item dialog box. Be sure to call the new class module pgTextField.cs and write the following code:

```
public class pgTextField : Events
{
        protected override void OnCreate(Container page)
        {
                base.OnCreate(page);
                Field dfField;
                dfField = page.Field("dfField");
                dfField.StrValue = "Text from code";
        }
}
```

Rebuild Tutorials.DLL and upload it using the BLA to the BLS. Check in the BLA Explorer view that the new class and method are visible.

In the Explorer view of i5 Studio, click on the Logic tab of pgTextField and associate the page with the Tutorials library and the pgTextField class. Previewing pgTextField will show that text has now been inserted into the dfField data field.



Creating new objects dynamically

Any object that can be created using i5 Studio can also be created dynamically with business logic code. To create a new object, simply make a reference to an object that does not already exist. The position of an object created dynamically in this way depends on three things: whether or not the page on which it is created contains objects that have been created 'statically' in the Designer, the order in which dynamically created objects appear in your business logic code and whether or not any particular object has its Order property set. Objects created dynamically appear after the last static object unless the Order property has been set. We look at the Order property shortly.

The following code creates an image object, for example, and it will be the last object on your page:

```
        using (Image myImage = pgUserDetails.Image("imgLogo"))
        {
                myImage.StrValue = "/i5/splash6.jpg";
                myImage.ImageType = ImageType.URL;
        }
```

In this example, *if* the imgLogo object has not been created previously in i5 Studio, it will be created when i5 processes the business logic code in which it is referenced. Some of the properties of an object that are created in this way will be set automatically to a default setting. Most objects however, will require some of their properties to be set in business logic code so as to be rendered in the required fashion.

The code sample above employed the C# keyword 'using'. Typically, the using keyword is employed as part of a 'directive' - as when you define a namespace for example. However, it can also be employed as part of a statement. When it is employed in a statement, it specifies that the object with which it is used has a defined scope (set by braces {}). Once the end of that scope is reached, the object will be disposed of. Effectively then the using keyword allows you to define an alias for an object, obviating the need to write out in longhand the identifier pgUserDetails.Image("imgLogo").

Adding the following code fragments to the code for a page OnCreate event will demonstrate the dynamic creation of some of the most common i5 objects. Try adding them, in order, to the pgTextField class you just created

Read-only text field

```
using( Field myField = page.Field("dfText"))
  {
        myField.TextSize = 2;
        myField.StrValue = "This is a demo";
        myField.Justify = Justify.Right;
        myField.Editable = false;
        myField.TextColorBg = "p3";
  }
```

You do not have to create a copy of the object within business logic, as we have done above. You can refer directly to a previously created object just as easily, as shown in the examples below. However, your code will generally be easier to manage if you do create copies of your objects – not least because you will be spared the need to write out the full reference to the object.

Radio-button group

```
page.RadioGroup("rgNew").Title = "Radio button group";
page.RadioGroup("rgNew").Lookup = "1,Adrian,2,George,3,Robin";
// alternatively, doing same thing with the using keyword
using (RadioGroup myRg = page.RadioGroup("rgNew"))
{
      myRg.Title = "Radio button group";
      myRg.Lookup = "1,Adrian,2,George,3,Robin";
}
```

List box

```
page.ListBox("lbNew").Title = "List box";
page.ListBox("lbNew").Lookup = "1,John,2,Dina,3,Alec";
```

Page link

```
page.PageLink("pbNew").DrawStyle = DrawStyle.TextualHyperLink;
page.PageLink("pbNew").StrValue = "pgHello";
page.PageLink("pbNew").Page = "AnotherTutorial.pgHello";
```

Image

```
page.Image("imgNew").ImageType = ImageType.URL;
page.Image("imgNew").StrValue = "/i5/palm.gif";
page.Image("imgNew").Title = "Picture";
page.Image("imgNew").Order = 1;
```

The last code fragment above, for an image object, explicitly tells i5 where in the order of objects created on a page the image should come – setting the object's Order property to 1 instructs i5 to make it the first object on the page, thus overriding the default order of creation. The Order property will override both the order of statically and dynamically created objects (so you could place the above code fragment at the end of business logic for a page which also has five statically created objects and the image will still appear first on the page).

Any public method/property of any of the objects/classes in the i5 API can of course be set on a page OnCreate event. For example, by default, an i5 page will have its layout method set to absolute positioning. This is not always advisable if creating pages and their components dynamically, because great care must be taken in specifying exactly where a component sits on a page when it is created. This again can create problems if the size of a component is determined dynamically at run time. The following code snippet will change the layout method of a page to flow-based layout upon an OnCreate event:

```
protected override void OnCreate(Container page)
{
    base.OnCreate(page);
    page.Layout = Layout.Flow;
    // etc – rest of code here
}
```

If you have worked through the *Getting Started* guide for i5, you will know that it is possible to set the tab order of components included in an i5 page by setting the Tab Index property in the Tab View of the Designer. As the Tab Index property is exposed in the API, it is of course also possible to set this property using business logic and this can be exploited to set the focus to a specific field on a page when it is created.

The basic rule for the setting of focus is that the component with the lowest tab index when the page is created gets focus. By default, the tab index of a component is correlated with the order in which components are created on a page, such that the first component on the page receives the lowest tab index value and thus will automatically receive focus.

In the code below, the tab ordering of three components on a page is explicitly set to run backwards – i.e. from highest to lowest. This determines that the last field created will receive focus:

```
protected override void OnCreate(Container page)
{
    base.OnCreate(page);
    page.Field("df1").Title = "Field1";
```

```
                page.Field("df1").TabIndex = 3;

                page.Field("df2").Title = "Field2";
                page.Field("df2").TabIndex = 2;

                page.Field("df3").Title = "Field3";
                page.Field("df3").TabIndex = 1;
        }
```

For further information regarding the properties/methods of i5 components refer to the *i5 API Reference Guide*.

Although some of the properties of some of dynamically created objects can be changed manually in i5 Studio, the settings as defined in business logic code will always override any such manual changes.

Handling user input

In addition to the manipulation and dynamic instantiation of objects on a page, it is likely that a significant feature of any of the business logic that will sit behind your i5 pages will be the processing of data entered into fields and other objects by end users. There are a number of ways in which user input can be handled by i5, perhaps the most obvious being through the use of the page OnSubmit event.

As mentioned earlier, the OnSubmit event is only triggered when a user clicks on an i5 page link button, typically at the end of a series of actions, and this is what makes it the appropriate event for invoking business logic to process data.

When the user clicks on a page link, all the values entered by the user on that page are sent back to i5 from the client browser for processing. Values returned for processing in this way may be accessed and modified in code for an OnSubmit event although manipulating the properties of any i5 object – e.g. the colour of a data field or the title of a list box – will have no effect at this point. This is because the page is not actually being created during the OnSubmit event and so is only available for reference purposes.

On completion of the OnSubmit event, i5 decides which page to create next and this is determined by the target page value of the page link button, as set in i5 Studio - unless overridden by any business logic code set on the OnSubmit event, directing the user to a different page.

More information on page redirection can be found in the tutorial dealing with this topic later in this chapter and in the Function Reference – see the NextPage property of the Session class.

Tutorial: Handling User Input

User input processing does not necessarily have to be carried out on the OnSubmit event. Indeed, a good deal of user input processing can be carried out without needing to use it at all, as the following simple exercise will demonstrate. The code creates a program that acts as a simple calculator, adding and subtracting numbers typed in by the user, and it works entirely on the page OnCreate event.

In i5 Studio, do the following:

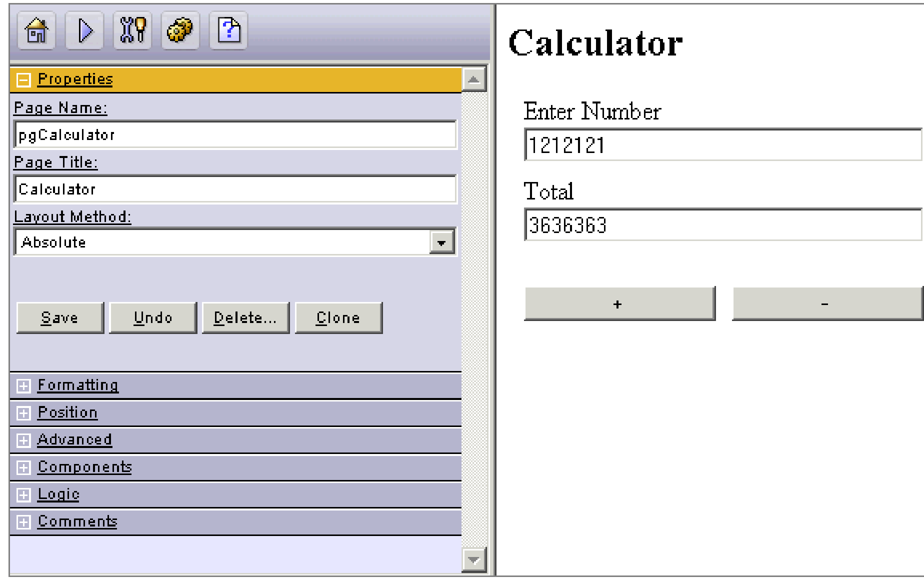
- Add a new section, TutorialThree and a new detail page, pgCalculator, for that section.

- Add two data fields to the page: dfNumber, with the title Enter Number, and dfTotal with the title Total.
- Add a Page Link button, pbAdd and set its page to TutorialThree.pgCalculator. Make the text for pbAdd a plus symbol (+).
- Add a second Page Link button, this time called pbSubtract and position it next to pbAdd.

In its current state this page will not, obviously, perform any calculations. In order to rectify this some business logic is required. So, in Visual Studio, you should now add a new class to the Tutorials project, called TutorialThree, and add event handling code for the OnCreate event of pgCalculator as per the following code fragment:

```
public class pgCalculator : Events
{
	protected override void OnCreate(Container page)
	{
		base.OnCreate(page);
		Field dfNumber, dfTotal;
		dfNumber = page.Field("dfNumber"); // Get number entered by user
		dfTotal = page.Field("dfTotal");	// Get running total
		if (page.LinkPressed == "pbAdd")
		{	// user wants to add numbers so add them
			dfTotal.NumValue += dfNumber.NumValue;
		}
		else if (page.LinkPressed == "pbSubtract")
		{	// user wants to subtract number so subtract it
			dfTotal.NumValue -= dfNumber.NumValue;
		}
	}
}
```

Rebuild the DLL and upload the Tutorials library again. Having done that you will need to associate the business logic that you have written with the pgCalculator page in the Designer. To do this, navigate to the Logic tab of pgCalculator in the Tab view of the Designer. Locate the Tutorials dll via the dropdown list of dll files under the 'Library' heading and then select the 'pgCalculator' class from under the 'Class' heading. Be sure to save your changes. You should now find, when you preview the page, that you have a functioning (if rather crude) calculator.



Tutorial: Validating User Input

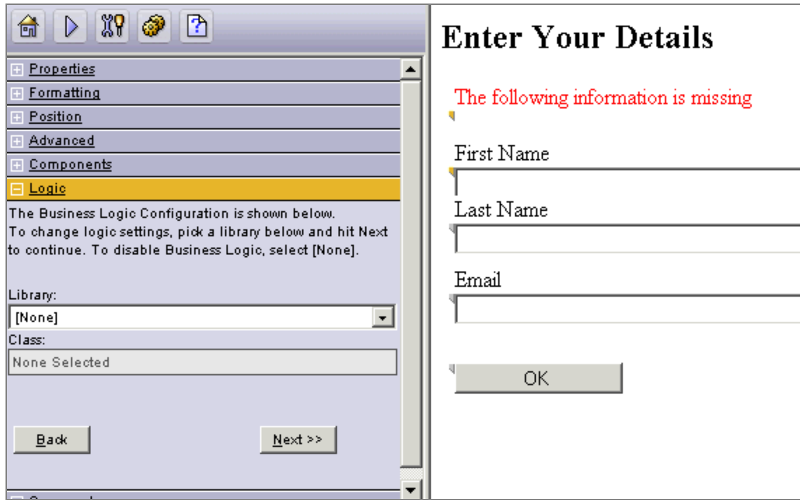
Processing user input on an OnCreate event, however, is not necessarily the most effective way of dealing with data. In particular, validating data can become rather fiddly. The OnSubmit event facilitates processing tasks such as data validation, so here we will show how to capture user input as it is submitted to i5 and to check that it is correct.

In this particular example, if the input is not correct or is missing, the same page will be displayed with an error message showing the names of the incorrect fields. If the input is correct the message displaying the missing data will be hidden - i.e. made invisible. The OnSubmit event will be used to capture the input as it is submitted to i5 and the OnCreate event used to hide the error message.

In i5 Studio, do the following:

- Add a new section called TutorialFour and a new detail page called pgValidation with the title 'Enter Your Details'.
- Add a data field to pgValidation called dfErrorText with the title 'The following information is missing'. On its Property tab make it a read-only by unchecking the Editable checkbox. On its Formatting property tab turn off its border, set its background color to transparent, and its text and title font color to red.
- Add data fields called dfFirstName, dfLastName, dfEmail with the titles First Name, Last Name and Email.
- Add a page link button called pbOK, make its text OK and set its Go to Page list box to TutorialFour.pgValidation.

The page should look something like this:



Add a class called pgTutorialFour to the Tutorials project in Visual Studio and use the following code for its OnSubmit event:

```
public class pgTutorialFour : Events
{
        protected override void OnSubmit(Container page)
        {
                base.OnSubmit(page);
                if (page.LinkPressed == “pbOK”)
                {       // validate user input
                        string sErrorText;
                        sErrorText = "";
                        if (page.Field("dfFirstName").StrValue == "")
                        {
                                sErrorText = "FirstName" + "\n";
                        }
                        if (page.Field("dfLastName").StrValue == "")
                        {
                                sErrorText += "LastName" + "\n";
                        }
                        if (page.Field("dfEmail").StrValue == "")
                        {
                                sErrorText += "Email" + "\n";
                        }
                        page.Field("dfErrorText").StrValue = sErrorText;
                }
        }
}
```

Rebuild the DLL, upload it and associate the business logic with the correct section in the i5 book.

In its current state this business logic will not hide the error text field when there is no actual error. To rectify this, the following code, written for the OnCreate event of the class pgValidation, allows the user to set the properties of the page objects.

```
protected override void OnCreate(Container page)
{
        base.OnCreate(page);
        Field dfErrorText = page.Field("dfErrorText");
        if (dfErrorText.StrValue == "" )
        {       // hide the error text box if no error text
                dfErrorText.Hidden =  true;
        }
}
```

Using the OnRequest event

Every time a user goes to an i5 page or an i5 page is refreshed, the page OnRequest event is triggered. The OnRequest event takes place immediately prior to the OnCreate event. The OnRequest event exists to allow i5 to request access to a page, and it allows business logic to be written to in order to redirect i5 away from the page that the event is triggered for - i.e. to a different page.

This is useful for security purposes as it can be used to steer unauthorized users away from specified pages. To explore the value of the OnRequest event, we will look briefly at page redirection.

It is worth noting that because the request event is triggered prior to the creation of the page, there are no objects on the page to modify. Equally, the developer cannot insert objects into the page, because at this stage, prior to the OnCreate event, the page simply doesn't exist.

Tutorial: Page Redirection

Trapping the OnRequest event for a particular page allows the developer programmatically to determine whether or not that page should indeed be created. Business logic code can, in these circumstances, allow the developer to re-direct requests for a page to somewhere completely different. The value of this in relation to security issues, for example, is self-evident. Using a combination of OnSubmit and OnRequest events, this tutorial will illustrate how users can be redirected away from a specific page.

To do this, a main page which allows the user to choose a target page from a list box will be created. And this will use business logic written on the OnSubmit event to capture the value selected. Next, some additional code will be written on the page OnRequest event for one of the possible target pages to ensure that when that page is requested, the user is not permitted to view it but is instead directed elsewhere.

In i5 Studio, do the following:

- Create a new section in your Tutorials book in i5 Studio, calling it TutorialFive and add a new detail page called pgMain, giving it a title something like Choose A Page.
- Add three detail pages, called pgOne, pgTwo and pgThree and titled Page One, Page Two and Page Three respectively.

- To pgMain add a new List Box, called lbLocation and select the Static Content option for it, so that you can type in the list values yourself. Type in these values for the list: pgOne,Page One,pgTwo,PageTwo,pgThree,Page Three. (List boxes with static content require you to enter, sequentially, a comma-separated list of values to be stored and values to be displayed).
- Add a page link button to pgMain, called pbGo, make its text 'Go and set its 'Go to Page' value to TutorialFive.pgMain. Note that without any business logic, the Go button on pgMain will simply redirect the user back to itself.

Create a new class, TutorialFive in the Tutorials project and add the following event handling code:

```
public class TutorialFive : Events
{
        protected override void OnRequest(Container page)
        {
                base.OnRequest(page);
                if (page.Name == "TutorialFive.pgThree")
                {
                        page.Session.NextPage = "TutorialFive.PgOne";
                }
        }
        protected override void OnSubmit(Container page)
        {
                base.OnSubmit(page);
                if (page.Name == "TutorialFive.pgMain")
                {
                        string sLocation;
                        sLocation = page.ListBox("lbLocation").StrValue;
                        page.Session.NextPage =  "TutorialFive." + sLocation;
                }
        }
}
```

Rebuild the DLL, upload the library and associate the TutorialFive class from the Tutorials library with TutorialFive.pgMain and also with TutorialFive.pgThree.

Note how our code checks the name of the page at the start of each method. This is required because the business logic is being associated with two pages in the TutorialFive section and we wish to be sure that the correct actions are executed for the correct pages. For example, if we did not check the page name in the code for the OnRequest event, the user would never get to the Main page!

To make this code of practical value, of course, we would need to do a little work on it. If every user were directed away from a page in an application, there would be little point in keeping the page. However, it is not difficult to imagine how you might run some checks on the user's identity when they request a page, in order to determine whether or not they would have permission to access it. To do this you would probably want to make use of i5 global variables, which are discussed in the next chapter. In this chapter we have restricted our attention to the specific types of event to which i5 will respond and how these events may be exploited programmatically.

3. Managing User State using Container Objects

In the previous chapter we looked at page events. Events in i5 are triggered for container objects. In addition to 'containing' the objects which are eventually created on a page and allowing the developer to create and to modify such objects dynamically, containers also permit access to useful data about things such as a client's session connection. In this chapter we take a look at how to manage user state using containers and so to deal adequately with the typically stateless nature of web pages.

User State Management

Using the Container passed from i5, information relevant to a client's session connection can be accessed. The *Session* object which sits in the container provides access to a number of properties to get or set. For example, the following code displays the syntax for returning a *Browser* object:

```
Container.Session.Browser()
```

This fragment will return a *Database* object:

```
Container.Session.Database()
```

Whilst this fragment will return a user identity:

```
Container.Session.Sqluser()
```

More generally, Session allows the developer to get and to set state information for CGI parameters, along with a number of other elements – cookies, for example. The following code fragment shows how cookies might be used to create a simple persistent client-side state:

```
page.Session.Cookie("visited_before") = "yes";
```

Refer to the *i5 3.5 .Net API Reference* for further information on the Session object.

State variables

State variables are valuable because they allow information to be held so that it can be reused within the same piece of business logic or, perhaps, in another piece of business logic.

State variables in i5 hold values on a per-user basis and can be instantiated as numbers, dates, binary and strings, or a single-dimension array of any of these. The contents of these variables can be retrieved and set in code. In i5 there are two types of state variables: *Global* and *Page* variables.

Global variables

Global variables are by definition in scope for all of the business logic code for a given user connection and for the lifetime of that connection. This means that the contents of any particular Global variable is likely to be different for every user connected to i5.

Global variables are useful for holding on to information about a user or session, or for sharing information over a number of pages. For example, when a user logs in his/her username and access level can be stored for retrieval later on.

The following code fragment shows how:

```
page.Session.GlobalVar("varUserName") = dfUserName;
page.Session.GlobalVar("varUserAccess") = "restricted";
```

Page variables

Page variables are similar to their Global cousins in that they exist only on a user connection basis, but dissimilar in that they are only in scope for the page on which they were created. In other words, different pages can have Page variables of the same name, but they will contain different values.

The following code fragment, for example, records how many times a user has visited the current page:

```
int nCount;
nCount = page.Session.PageVar("varCounter");
page.Session.PageVar("varCounter") = nCount + 1;
```

To get or set a Page variable which is out of scope, i5 needs to know the name of the section that the page lives in, followed by the name of the page itself:

```
page.Session.PageVar("mySection.myPage.myPageVar")
```

As an example, imagine creating a New Invoice Wizard page when a user chooses a company to invoice from a list of companies. During the submit event for the page, the submitted value from a Company list box could be retrieved and used to set a CompanyId variable on the Invoice page to the chosen company:

```
int nCompanyId;
nCompanyId = lbCompany.NumValue;
page.Session.PageVar("Invoicing.pgInvoice.varCompanyId") = nCompanyId;
```

Having set the Page variable on the Invoice page to the selected value it is then possible to retrieve that value during the create event for that page, using it to get the details of the company from a database table:

```
public class pgInvoice : Events
{
    protected override void OnCreate(Container page)
    {
        base.OnCreate( page );
        //get back variable
        int nCompanyId;
        nCompanyId = page.Session.PageVar("varCompanyId");
            ...rest of code here
    }
}
```

Initialisation of Page Variables

Before we have a more detailed look at how page variables can be used in an application, there is a need to address how Global and Page variables behave when the code attempts to get a previously undefined variable. In i5 3.0, using COM-based languages GlobalVar() and PageVar() both return Variants. In i5 3.5, using .NET languages, the 'equivalent' is the Object datatype (object in C#). In COM the Variant data type is the data type for all variables that are not explicitly declared as some other type, and in Visual Basic this means any variable which is declared without being set to a particular type such as Integer will be set as a Variant. Likewise in .NET, the Object data type is the data type for all variables not declared as some other type.

In the following code snippet, which you should try for yourself, we use an uninitialised Global variable on a page to try to set the value of a variable of type object. As the effect of the code suggests, there is no default setting for a global or a page variable: retrieving a variable which is not initialised results in the object's type being set to whatever is in the heap. See what happens when, having navigated away from the page you then come back to it.

```
public class ObjectDemo : Events
{
        protected override void OnCreate( Container page )
        {
                base.OnCreate(page);
                object myVal = new object();
                myVal = page.Session.GlobalVar("initialised");
                if (myVal == null)
                {
                        page.Text("info").StrValue = "GlobalVar() not initialised";
                }
                else
                {
                        page.Text("info").StrValue = "GlobalVar()  initialised";
                }
                page.Session.GlobalVar("initialised","Now initialised");
        }
}
```

Tutorial: Building an input wizard using state variables

Some of the properties of state variables can be demonstrated by building a simple input wizard to collect user information through consecutive input pages and then to display the information collected. The wizard works by storing the data collected on each page into global variables and then retrieving it at the end of the collection process.

To build this wizard several pages are required, for collection of information, display and resetting. Business logic also needs to be written to gather up the information into some state variables for later access.

So, in i5 Studio, do the following:

- In a new section – called TutorialSix**,** add three detail pages: pgOne, pgTwo and pgFinished. Give the pages the titles: Enter Your Name, Enter Your Address and Finished respectively.
- Add the input fields dfFirst and dfLast to pgOne, dfAddress and dfPostCode to pgTwo
- Add a Page Link called pbNext to pgOne with its text set to "Next >>" and its target page to TutorialSix.pgTwo. pgOne should now look something like this:

- Add two Page Links to pgTwo, one called pbBack, with its text to “<<Back” and its target page to TutorialSix.pgOne and a second called pbNext, with its text set to “Next>>” and its target page to TutorialSix.pgFinish. pgTwo should now look something like this:

- Add a Page Link called pbBack to pgFinish with its text set to <<Back and its target page to TutorialSix.pgTwo. pgFinish should now look something like this:

At the moment these pages are completely stateless: anything entered in the data fields on any of these pages is lost as the user navigates away from them. The next step is therefore to write some business logic to allow us to capture any information entered into a set of state variables.

Add a new class to the Tutorials project in Visual Studio, called TutorialSix and add the following code for the submit and create events for pgOne:

```
public class pgOne : Events
{
        protected override void  OnSubmit( Container page )
        {       // store field values into variables
                base.OnSubmit( page );
                page.Session.GlobalVar("dfFirst", page.Field("dfFirst").StrValue);
                page.Session.GlobalVar("dfLast", page.Field("dfLast").StrValue);
```

```
        }
        protected override void OnCreate( Container page )
        {           //restore field values from variables
                    base.OnCreate( page );
                    page.Field("dfFirst").StrValue = (string)page.Session.GlobalVar("dfFirst");
                    page.Field("dfLast").StrValue = (string)page.Session.GlobalVar("dfLast");
        }
}
```

And the following code for the Submit and OnCreate events for pgTwo:

```
public class pgTwo : Events
{
        protected override void OnSubmit( Container page )
        {           // store field values into variables
                    base.OnSubmit( page );
                    page.Session.GlobalVar("dfAddress", page.Field("dfAddress").StrValue);
                    page.Session.GlobalVar("dfPostCode", page.Field("dfPostCode").StrValue);
        }
        protected override void OnCreate( Container page )
        {           //restore field values from variables
                    base.OnCreate( page );
                    page.Field("dfAddress").StrValue = (string)page.Session.GlobalVar("dfAddress");
                    page.Field("dfPostCode").StrValue =  (string)page.Session.GlobalVar("dfPostCode");
        }
}
```

After rebuilding the library, upload it and associate it with the appropriate pages in the section TutorialSix. You should how find that state is being retained as you navigate from page to page. You may find that objects on the pages overlap – if they do, that is probably because you have the Layout Method set to Absolute, so you will probably want to change that (as we have done in the code below – see also the discussion earlier in this chapter). Note the way that in the code samples above we have cast the global variables using the (string) notation – we are obliged to do this because, as we mentioned earlier, global and page variables are considered to be variables of type 'object' in .Net, and so there will be a type mismatch if we wish to assign the values that they contain to a string, unless we change the object's type.

Some business logic is still needed to extract the values from the state variables so as to display them on the Finished page. Create the following class for the page pgFinish.

```
public class pgFinish : Events
{
        protected override void OnCreate( Container page )
        {
                    base.OnCreate( page );
                    page.Layout = Layout.Flow;
                    using ( Field myField = page.Field("dfDetails"))
                    {
                                myField.Title = "Your details are";
```

```
                myField.TitleBold = true;
                myField.StrValue =  (string)page.Session.GlobalVar("dfFirst") + " "
                                        + (string)page.Session.GlobalVar("dfLast") + "\n"
                                        + (string)page.Session.GlobalVar("dfAddress") + "\n"
                                        + (string)page.Session.GlobalVar("dfPostCode");
                myField.Editable = false;
                myField.Border = LineStyle.None;
                myField.Order = 1;
            }
        }
}
```

The page should now look something like this:

It would be helpful if the details for every new user could be reset. This can be done by creating a new page to act as a starting point for new users.

On the OnRequest event for the starting page, the user details stored in the state variables will be reset and the user redirected to pgOne.

Add a new detail page to the section TutorialSix called pgNewUser. The page is not going to be displayed, so there is no need to add a title. However, we will use it to reset our global variables, so add the following code for the request event for pgNewUser to the TutorialSix class in Visual Studio:

```
public class pgNewUser : Events
{
        protected override void OnRequest( Container page )
        {
                base.OnRequest( page );
                page.Session.GlobalVar("dfFirst","");
                page.Session.GlobalVar("dfLast","");
                page.Session.GlobalVar("dfAddress","");
                page.Session.GlobalVar("dfPostCode","");
                // redirect user to the first page in section
                page.Session.NextPage = "TutorialSix.pgOne";
        }
}
```

Rebuild the library, upload it and associate the library and class with pgNewUser.

Add a page link to pgFinish and call it pbNew with ‘New’ for its text. Set its target page to pgNewUser. When this is pressed, the user will be taken to pgOne, with all the global variables reset and ready to accept details of another user.

Summary

Page and Global variables are an important element in the i5 developer's tool kit. They are especially handy for dealing with a problem that any developer of websites has to face sooner or later: the management of user state. In the course of this chapter, we've introduced you to the use of i5 page and global variables, showing you how you might use them to maintain state. In the next chapter we will move onto a consideration of working with databases.

4. Databases, connectivity and child containers

Clearly a crucial factor in the development of successful data-driven web applications is the interface which your application establishes with its data source or sources. i5 Studio permits a great deal of flexibility in dealing with the data used to drive the pages in an i5 application, but there will be times when you need to manipulate data programmatically using business logic. The API provides classes – Database and SQL – to establish database connectivity and a high degree of control over your data and ensures a consistent programming interface to a range of relational database management systems.

This chapter covers the functionality that these classes provide and examine the way that they may be used within child containers, another useful tool in the i5 developer's toolkit.

Database Connectivity

The connectivity classes

All database connectivity in i5 is handled by i5SQL processes. i5SQL processes are analogous in their functioning to processes based on the better known remote procedure call protocol. i5SQL processes replace the rpcSQL processes previously used in net.db. Microsoft no longer supports *rpc* and the change in i5 reflects this. i5SQL processes facilitate the execution of queries on a database from remote clients by providing a transport layer-neutral protocol for passing procedure calls with arguments to a server for processing. Whilst i5SQL processes will typically reside on the same server as the i5 server, they do not need to be on the same machine as the code executing business logic. The number of i5SQL processes which can run simultaneously in i5 is set by default at five, but can be changed when required. This provides true connection concurrency and permits more than one SQL request to be running at the same time against any i5 supported database.

Database

i5's *Database* class offers both *connectivity* to a named database and cursor pooling via the i5 *SQL* class. A connection to the Database pool creates a workspace from which many SQL handles can be obtained. The concept of the workspace forces all SQL handles within that workspace to be affected equally - i.e. a COMMIT/ROLLBACK statement will affect all the SQL handles in that workspace.

The following code fragment shows how to make use of the database pool used by i5 for the current user connection:

```
Database db;
db = page.Session.Database;
```

To access and modify data, SQL provides a series of methods to prepare and execute SQL queries, commits, rollbacks and so on. By using the Database.Connect method, one or many SQL class instances can be connected and the methods of the SQL class utilised.

SQL class instances connected via the same Database object share a cursor pool and are seen by the database as a single user, whereas SQL class instances connected via a different Database object have different pools and are seen by the database as different connections. By creating two or more Database objects business logic can be used to connect to a number of different databases simultaneously. The following fragments show how to connect to and disconnect from a database, to execute SQL and to fetch values.

To connect:

```
SQL connection;
connection = db.Connect();
```

To disconnect:

```
connection.Disconnect();
```

To execute SQL:

```
connection.Prepare(<SQL statement>);
connection.Execute( );
```

To fetch a value:

```
connection.FetchNext;
sFirstColumn = connection.StrValue(0);
sSecondColumn = connection.StrValue(1);
```

For further information on i5's database connectivity please refer to Appendix - Connectivity and Scalability.

Tutorial - Connecting to a database

To demonstrate how the API classes work in practice, here's an example showing how they are used in business logic:

In the Tutorials project, add a new class module called pgTutorialSeven and then add a method to it as follows. Note that we are using flow-based layout, as rather a large number of objects are going to be created on the page (you can experiment with the code to see what happens if it executes using absolute layout as well!!)

```
protected override void OnCreate( Container page )
{
        base.OnCreate( page );
        page.Layout = Layout.Flow;
        using ( SQL mySql = page.Session.Database.Connect() )
        {
                if( mySql.Prepare("select PRODUCT_ID,TITLE from PRODUCT" )==true)
                {
                        if ( mySql.Execute() == true  )
                        {
                                while (mySql.FetchNext() == true)
                                {
                                        page.Field("df" + mySql.StrValue(0)).Title = "Name";
                                        page.Field("df" + mySql.StrValue(0)).StrValue =
                                                mySql.StrValue(1);
                                }
                        }
                }
                mySql.Disconnect();
        }
}
```

Rebuild the DLL and upload it using the BLA. Note that you could just as easily have used the SQL.PrepareAndExecute() method, which is a new addition to the i5 API in version 3.5. Had you chosen to do so, you would have substituted the following code for the first two 'if' statements in the code above:

```
....previous code here
if (mySql.PrepareAndExecute("select PRODUCT_ID,TITLE from PRODUCT" )==true)
{
        while (mySql.FetchNext() == true)
        {
                ...rest of code here
        }
}
mySql.Disconnect
...etcetera
```

To continue the tutorial though, in i5 Studio you should now create a new section called TutorialSeven and add a new page called pgTutorialSeven, then upload the business logic and associate it with the page. The new page should now display a column of data fields for all the companies in the database.

The business logic accomplishes the following actions: On the page OnCreate event it creates an SQL object, connected to the current database on the current session. It then prepares and executes an SQL query to retrieve a result set. Note how the query must be defined as a string. In the While loop, the code then dynamically creates a data field for each data item in the result set, gives it a title and then inserts a value drawn from the result set into it.

As an example, this is fairly rough and ready code (there's no error handling, for example) but it does illustrate the key point, which is that SQL objects can be used very easily to populate pages with data at runtime.

Using Try/Catch/Finally

The example above shows you in a very simple way how to connect to and extract data from a database. When you are working with real-world applications, it is likely that you would want to wrap some error handling code around your business logic. Although we won't incorporate such code into the bulk of our examples in this guide, we will look here at how to use C# error handling functionality to make your data connections a bit more robust (you will need mentally to wrap our examples in your own exception handling code!).

The following bit of code illustrates how we might make this code a bit safer by incorporating a way to handle exceptions when working with databases:

```
protected override void OnCreate( Container page )
{
        base.OnCreate( page );
        SQL mySql = page.Session.Database.Connect();
        try
        {
                if( mySql.PrepareAndExecute("select PRODUCT_ID,TITLE from PRODUCT") == true)
                {
                        while( mySql.FetchNext() == true )
```

```
                    {
                              page.Field("df" + mySql.StrValue(0)).Title = "Name";
                              page.Field("df" + mySql.StrValue(0)).StrValue = mySql.StrValue(1);
                    }
          }
          else
          {
                    throw new Exception(mySql.LastError.Text);
          }
          mySql.Commit();
     }
     catch( Exception  ex )
     {
          mySql.Rollback();
          throw( ex );
     }
     finally
     {
          mySql.Disconnect();
     }
}
```

This code is pretty simple. Basically, by using the try loop we are giving ourselves the opportunity to catch any problems that might arise from connecting with the database – and throwing an exception accordingly. Putting the .Disconnect method in the finally statement ensures that whatever happens in our try loop, we will always disconnect from the database. If we were writing to the database, we might want to make use of the SQL.Commit and SQL.Rollback methods – committing our changes at the end of the try loop and rolling back any changes if an exception is thrown.

To verify that this sort of coding construct works, try altering the code as follows. Change the SQL statement in the 'if' test to read as follows. The database server will object strongly to the use of 'frim' instead of 'from':

```
if( mySql.PrepareAndExecute("select PRODUCT_ID,TITLE frim PRODUCT") == true)
```

Once you have done that, rewrite the 'catch' code block so that if an error occurs you display an error message on the page.

```
catch( Exception  ex )
{
    page.Field("dfError").StrValue = "error thrown in SQL" + ex.Message;
    mySql.Rollback();
}
```

In the real world, you would probably not handle your exceptions by displaying such an unhelpful message on the page where the problem is occurring – you would want to use a more

sophisticated and helpful message. However, the point here is simply to show you how the construction works.

Tutorial - Creating a child table dynamically

The previous example shows one way in which data from a database can be retrieved and displayed in i5. A more effective way to do this in i5 is to use a *child table*. Unlike other i5 page objects - a field or a radio button, for example - a child table is a container and can therefore be used to hold objects created in the same way as the objects that are created in a page container. Objects in a child table will behave in the same way as objects on a page – they can be ordered in the same way, have their colouring or alignment changed and so on.

To explore the use of child tables, we will look at how to create one which displays data from several tables in the i5demo database.

Create a new section in i5 Studio, called TutorialEight and a new, but untitled, detail page called pgTutorialEight.

In Visual Studio add a new class module to the Tutorials project, call it pgTutorialEight and add the following event handling code for its create event:

```
protected override void OnCreate( Container page )
{
        base.OnCreate( page );
        // create a child table
        using ( Container myTable = page.Table( "tblOrders" ) )
        {
                myTable.DataSource = "dbo.[ORDER] o, dbo.PRODUCT p";
                myTable.Title = "Orders";
                myTable.Join = "o.PRODUCT_ID=p.PRODUCT_ID";
                myTable.SortBy = "p.TITLE asc";
                myTable.Dimension.Width = 400;
                myTable.Dimension.Height = 0;
                myTable.FixHeight = false;
                myTable.Coordinate.Y = 50;
                myTable.Border = LineStyle.None;
                // assign results set to data fields
                myTable.Field("dfOrderNo").Column = "o.ORDER_CODE";
                myTable.Field("dfOrderNo").Title = "Number";
                // product name
                myTable.Field("dfName").Column = "p.TITLE";
                myTable.Field("dfName").Title = "Name";
                myTable.Field("dfName").DataType = DataType.String;
                // product price
                myTable.Field("dfPrice").Column = "o.COST";
                myTable.Field("dfPrice").Title = "Cost";
                myTable.Field("dfPrice").Justify = Justify.Right;
                myTable.Field("dfPrice").Picture = "0.00";
                myTable.Field("dfPrice").DataType = DataType.Number;
```

```
        }
    }
```

Rebuild the DLL, upload the library and associate the new class with pgTutorialEight. Note the use of compound expression for the invoice amount column.

The new page should show all the orders in the Order table along with the cost of each order and the name of the product ordered, taken from the Product table:

Number	Name	Cost
1231	Acqua di Gio for men	29.00
6548659	Armani for men	60.00
43648893	Armani for men	90.00
5437658765	Baby Rose Jeans	30.00
201	Black Jeans for men	9.00

Tutorial: Filtering a result set

When dealing with large amounts of data it is preferable to be able to filter the results of a query before the results are rendered on the page. In this instance we can do this by giving the user the means to select a company from a populated list box and then using the value returned as a bind variable for a child table *Query* property.

In the same C# class as in the previous tutorial:

Modify the create event for pgTutorialEight by inserting the following code to create a list box and a button to refresh the page:

```
protected override void OnCreate( Container page )
{
        base.OnCreate( page );

        // add a dropdown list of companies
        page.ListBox("lbCustomer").Title = "Choose Customer";
        page.ListBox("lbCustomer").Lookup = "SQL:SELECT CUSTOMER_ID, COMPANY_NAME " +
                                            "FROM dbo.CUSTOMER ORDER BY COMPANY_NAME asc";
        // add a refresh button
        page.PageLink("pbRefresh").StrValue = "Refresh";
        page.PageLink("pbRefresh").Align = Align.Right;
        page.PageLink("pbRefresh").Coordinate.X = 260;
```

Modify the code to build the child table as follows. Note the use of the .Query property to set the 'Where' clause for the SQL statement i5 will use to query the database.

```
        // create a child table
        using ( Container myTable = page.Table( "tblOrders" ) )
        {
                myTable.DataSource = "dbo.[ORDER] o, dbo.PRODUCT p";
                myTable.Title = "Orders";
```

```
                myTable.Join = "o.PRODUCT_ID=p.PRODUCT_ID";
                myTable.SortBy = "p.TITLE asc";
                // add query property
                myTable.Query = "o.CUSTOMER_ID = :lbCustomer";
                myTable.Dimension.Width = 400;
                myTable.Dimension.Height = 0;
                myTable.FixHeight = false;
                myTable.Coordinate.Y = 50;
                myTable.Border = LineStyle.None;
                // assign results set to data fields
                myTable.Field("dfOrderNo").Column = "o.ORDER_CODE";
                myTable.Field("dfOrderNo").Title = "Number";
                // product name
                myTable.Field("dfName").Column = "p.TITLE";
                myTable.Field("dfName").Title = "Name";
                myTable.Field("dfName").DataType = DataType.String;
                // product price
                myTable.Field("dfPrice").Column = "o.COST";
                myTable.Field("dfPrice").Title = "Cost";
                myTable.Field("dfPrice").Justify = Justify.Right;
                myTable.Field("dfPrice").Picture = "0.00";
                myTable.Field("dfPrice").DataType = DataType.Number;
        }
```

Rebuild the DLL and from the BLA, upload the library.

Choosing a company from the list box on pgTutorialEight and refreshing the page will cause the child table to display only those invoices belonging to the company selected.

The layout method for this page has been set to flow-based. When creating multiple components on the fly, absolute positioning will create components directly next to each other so use flow-based layout to ensure that objects appear with an orderly distance between each other.

i5 Child table events

As shown in previous chapters, i5 container objects have their own events and can have business logic assigned to them. For this reason all container objects have a Logic properties tab in i5 Studio. Child containers - containers within containers such as tables and groups - are no exception to this general rule. There are four types of event for child containers and they are, in order of execution, *Construct, OnCreate, Submit* and *SubmitLast* events.

OnConstruct — This type of event provides a convenient point at which to construct all the SQL-aware columns in a Child Table or Group. The event is triggered just before the container is populated with data from the SQL database. All object properties and object values in the container can be modified at this point although the object values themselves may be overwritten if later populated with data by i5.

OnCreate — The OnCreate event is triggered just before the container is drawn by i5 and

	just after it has been populated with data from the database. All objects and object values in the container can be modified at this point.
OnSubmit	The OnSubmit event is triggered when a user clicks on any i5 Page Link button, including page links which are not owned by the container. On submission, values in the container can be accessed and modified before the container is processed by i5. The properties of the objects in the container can be accessed at this point although changing their value would be meaningless as the page is historical and cannot be changed
OnSubmitLast	The OnSubmitLast event is identical to the OnSubmit event apart from the fact that the OnSubmitLast event is called after all the OnSubmit events. This can be useful for building plug-in components where all other code on the page has to be executed before the component performs its operation. An example of a component which uses this is the MultiTableUpdate component documented in the *Getting Started with i5* manual. MultiTableUpdate saves the data on the page during the OnSubmitLast event, after all child containers have received their OnSubmit.

The submit events of all containers are executed before the final submit event of the Page.

Writing business logic code for i5 child container events

Notation

As all child tables and groups can be assigned their own business logic class, the developer must take care to be sure that their code makes clear what object it refers to. In previous versions of i5, the event method was pre-prended by the object name. In i5 3.5 we recommend that you name your classes in such a way as to be able to make clear what kind of objects they are being used to instantiate. Where a child container has been assigned its own class, the notation for a child container event method is simply the event name. For example:

```
public class MyTable : Events
{
        protected override void OnCreate( Container table )
        {
                base.OnCreate( table );
                // event handling code for event goes here
        }
}
```

The method is passed an i5.Container class as a parameter. This class represents the contents of the container and is used to access and modify the properties of the child container's objects.

Tutorial: Accessing data in a child table

We just saw how to add a child table to a page during the OnCreate event of that page. The child table had a DataSource - which contained the names of the tables the data was to come from - and also its own data field objects. There is no necessary correspondence between the names of these data fields and the column names of the table to which they correspond – where the wizards in i5 automatically assign a name to a column in a child table, when programming using business logic, it is the developer's responsibility to assign appropriate names to data fields.

Let's look now at how the user can access and modify the data in a child table immediately before it is drawn. To do this, TutorialEight will be extended by adding an accumulating *Total Cost* column to the invoice list and a total balance displayed beneath the table.

In Visual Studio, modify the OnCreate event code for pgTutorialEight by adding a "Total" field dfTotal to the page

```
            ...
                // product name
                myTable.Field("dfName").Column = "p.TITLE";
                myTable.Field("dfName").Title = "Name";
                myTable.Field("dfName").DataType = DataType.String;
                // product price
                myTable.Field("dfPrice").Column = "o.COST";
                myTable.Field("dfPrice").Title = "Cost";
                myTable.Field("dfPrice").Justify = Justify.Right;
                myTable.Field("dfPrice").Picture = "0.00";
                myTable.Field("dfPrice").DataType = DataType.Number;
            }
            // add a new grand total field to the page
            using( Field dfTotal = page.Field("dfTotal"))
            {
                    dfTotal.Title = "Total Cost";
                    dfTotal.Picture = "0.00";
                    dfTotal.Editable =  false;
                    dfTotal.DataType = DataType.Number;
                    dfTotal.Coordinate.Y = 100;
            }
        }
```

Rebuild the DLL and from the BLA, upload the library. Previewing pgTutorialEight should show that a *Total Cost* field appears on the page. Of course, to make this field useful, some code is required to go through the table data, accumulate the order totals and fill in the dfTotal field below the table.

In Visual Studio, add the event handling code for the create event for tblOrders, as follows:

```
public class tblOrders : Events
{
        protected override void OnCreate( Container table )
        {
                base.OnCreate( table );

                table.Field("dfBalance").Justify = Justify.Right;
                table.Field("dfBalance").DataType = DataType.Number;
                table.Field("dfBalance").Picture = "0.00";
                double nBalance = 0.0;
                int nRow;
                nRow = table.RowTop;
```

```
            do
            {       // accumulate total cost of orders for this customer
                    nBalance +=  table.Field("dfPrice").NumValue;
                    table.Field("dfBalance").NumValue = nBalance;
                    //set index to point to next row
                    nRow += 1;
                    table.Row = nRow;
            } while (nRow <= table.RowBottom);
            // put total onto field on page
            table.Parent.Field("dfTotal").NumValue = nBalance;
        }
}
```

Rebuild the DLL and from the BLA, upload the library. We have hit a small snag. We have written code to dynamically instantiate a table on the page OnCreate event – that's not a problem. However, we do not have a table on the page that we can bind the business logic too.

There are two options here. The easier option is to add a "placeholder" child table to the page in the i5Studio IDE and then associate the tblOrders logic with it (on the Logic tab). If you do this, preview pgTutorialEight and select a company, you should now find that the page displays with the accumulated total balance to pay on its orders appearing at the bottom of the page.

Note the way that the table's own OnCreate event is used to loop through the rows in the table using the i5.Container.RowTop method to initialise a row counter variable. Note also the way that the parent property of the the table Container object is used to access objects on the parent of the table container. The parent in this case, of course, is the page itself, which in addition to containing the child table also contains the data field for the total balance for all orders.

There is another way in which we can trigger code to create a child table on a page. In the first instance, we would write out the logic for the table in an OnConstruct event, rather than an OnCreate event (this is because of the order in which i5 executes the SQL to populate a data-aware object), and then we would make a call to the object using the following syntax:

```
page.Table("myTableObjectName", "myTableObjectClass");
```

The first parameter in this call creates a name for the object, the second parameter provides the name of the class that you have written containing the OnConstruct event for the table object. Of course, it need not be a Table container that you create in this way – the syntax works for Group objects as well. This way of creating child containers dynamically does present some advantages, in so far as it allows you to create all your page objects at runtime rather than statically when you are designing your pages.

Tutorial: Editing and saving data in a child table

To complete this run-through on writing code for child tables, let's look at how the end-user can be allowed to edit selected columns in the table and then to save the edited data back to the database. We will do this here by adding an editable Complete Date column to our child table and a Save button

The first thing to be done is to modify the OnCreate event code for pgTutorialEight by adding a Complete Date column, and by adding a new "Save Changes" page link pbSave to the page

```
            ...
            //new column
```

```
myTable.Field("dfComplete").Column = "o.COMPLETE_DATE";
myTable.Field("dfComplete").Title = "Completed";
myTable.Field("dfComplete").Editable =  true;
myTable.Field("dfComplete").Border = LineStyle.None;
myTable.Field("dfComplete").Dimension.Width = 80;
myTable.Field("dfComplete").Justify = Justify.Right;
myTable.Field("dfComplete").Picture = "dd/MM/yyyy";
myTable.Field("dfComplete").DataType = DataType.Date;
...
// add a save button to parent page
```

After rebuilding the DLL and uploading the library from the BLA, check that the dfComplete field is now editable and that a "Save Changes" button appears on the page. Some code is now required to save the contents of the Complete column back to the database when the Save Changes button is pressed. This will be done using the page OnSubmit event as the values of edited fields can be captured here.

Add the following OnSubmit event code for tblInvoice to the business logic class pgTutorialEight:

```
protected override void  OnSubmit( Container table )
{
        base.OnSubmit( table );
        if (table.Session.LinkPressed == "pbSave")
        {
                SQL mySQL = table.Parent.Session.Database.Connect( );
                int nRow = table.RowTop;
                do
                {       // save the paid amount on this row
                        table.Row = nRow;
                        if (mySQL.Prepare("UPDATE [ORDER] SET COMPLETE_DATE = :val1"
                                + " WHERE ORDER_CODE = :val2") ==  true)
                        {
                                mySQL.StrBind( "val1", table.Field("dfComplete").StrValue );
                                mySQL.StrBind( "val2", table.Field("dfOrderNo").StrValue );
                                mySQL.Execute( );
                        }
                        // set index to point to next row
                        nRow += 1;
                } while (nRow <= table.RowBottom);
                mySQL.Commit( );
                mySQL.Disconnect( );
        }
}
```

As with the code in the previous tutorial, a row counter variable is used to loop through each row in the table. Each time the loop is run through an SQL update statement is prepared and

executed using the values in the dfComplete data field to update the Order.Complete_Date column in the database.

After rebuilding the DLL and uploading it, verify that pgTutorialEight works correctly by editing some completion dates and saving the changes.

Summary

In this chapter, some of the possibilities for manipulating data provided by the SQL and Database objects have been explored. The *i5 .Net 3.5 Reference* guide to the i5 API provides further information about the use of SQL and Database objects. It is worth consulting for details about the methods available to the developer when working with SQL and databases programmatically.

5. Building plug-in components

The i5 Net 35 API provides sufficient flexibility to allow you as a developer to create your own re-usable 'plug-in' components, which can then be added to the components menu in i5 Studio and used in exactly the same way as standard components.

From the point of view of i5, plugins are simply child containers whose appearance and behaviour can be defined and controlled by business logic, without this control affecting the parent container into which the component is plugged.

In this chapter, we will look at the processes involved in creating a plug-in component, making it appear in the components list in i5 Studio, and at the properties and the events to which such components will respond.

A new plug-in component

Creating the component

In this example, we will build a self contained component to output the old standby text "Hello World" and then we will configure it to appear on the list of available form components in the i5 Studio 'Add Component' explorer.

In the Designer add a section called TutorialNine and then a new detail page called TutorialNine. Add a group to the page TutorialNine and call it HelloWorld. You do not need to give it a title. Notice that the page is empty. This is because the group HelloWorld is empty.

In Visual Studio, add a new class, HelloWorld to your Tutorials project. The syntax for the OnCreate event function name is identical to the syntax for page OnCreate events. The function name is constructed from the name of the event, and the group's Container passed in as a parameter. For example:

```
public class HelloWorld : Events
{
    protected override void OnCreate(Container group)
    {
        // code for group component here
    }
}
```

Note that the Container passed into a group OnCreate event represents the group and the group contents, in the same way that the Container passed into the OnCreate event for a page event represents the page and the page contents.

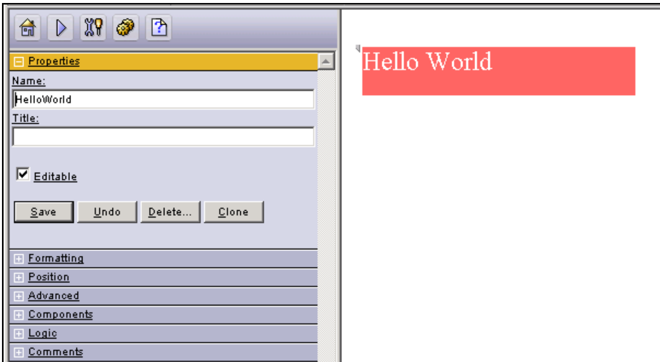
Add the following code to the OnCreate event of the HelloWorld class:

```
protected override void  OnCreate(Container group)
{
    base.OnCreate( group );
    using( Text txtHello = group.Text("textHello"))
    {
```

```
                txtHello.StrValue = "Hello, World";
                txtHello.TextColorFg = "255,255,255";
                txtHello.TextColorBg = "255,100,100";
                txtHello.TextSize = "+2";
        }
}
```

After saving the changes to HelloWorld.cs and recompiling the Tutorials project, you need to assign the HelloWorld class to the group placeholder that we created in the Designer. This process is identical to that for assigning business logic to a page, except that you select the HelloWorld class from the Logic tab of the HelloWorld group in the Designer

Notice that once the business logic is assigned to the group, the **HelloWorld** placeholder displays the text "Hello World".



Making the new component accessible from the Component list

The crucial thing about plugin components is that they can be added to pages in the same way as any other i5 component at design time. To make this possible though, we have to make any new component appear on our design time 'Add Component' list, something we accomplish by adding a configuration function to the business logic of the component. This function is used to provide extra configuration information about the class to i5.

Add the following code to your HelloWorld class:

```
protected override void OnQueryPlugin( PluginAttributes config )
{
        config.Type = PluginType.Group;
        config.Description = "My first plugin component";
        base.OnQueryPlugin( config );
}
```

On the File menu select Make Tutorials.dll and save (overwriting) the Tutorials.dll in the i5 'Lib' directory. i5 will automatically reload the DLL now that it has changed.

Clicking on New Component in the i5 Explorer beneath TutorialNine will show that the new Hello World component has been added to the component list. The component can now be

used in the same way as any other. Remember, though, that as the HelloWorld component, like every other plug-in, is built using business logic, to make it available to the developer, the Business Logic Server must be up and running.

By default, plugin components have the dimensions of 360 pixels by 200 pixels. You can alter this programmatically. For example:

```
protected override void OnQueryPlugin( PluginAttributes config )
{
    config.Type = PluginType.Group;
    config.Description = "My first plugin component";
    config.Dimension.Width = 500;
    config.Dimension.Height = 400;
    base.OnQueryPlugin( config );
}
```

Configuring components with properties

Whilst a reusable, self-contained component can be a very useful addition to the developer's toolkit, it is likely that on many occasions it will need to interact with the page in which it is situated or with the application in which it resides. By assigning properties to the classes used to build components, the developer can create components which are user-configurable in exactly the same way as any other i5 component.

A reusable email form, for example, is clearly a component that needs to be configurable: if it was forced to send emails to the same person every time an instance of it was used, it would not be particularly useful. However, if an 'Email To' property were available to configure the component's target email and an 'Email Host IP' property were available to configure which mail server were to be used to relay the email, the component would become considerably more valuable. Let us have a look at how we can create a component with configurable properties.

Creating a plug-in email form

A simple email entry form allows us to demonstrate the process of creating a component with user-configurable properties. The email form will allow a user to type in and send a message which will be sent to the email address configured on the component's Properties page. Note that this tutorial uses the i5 Mail object, as you will see in the code shortly.

In i5 Studio add a new section called TutorialTen and then a new detail page called TutorialTen.

Add a new Group to the page TutorialTen and call it SimpleEmailSender. Set its title to "eMail Form". This group will be used as a placeholder for our component.

Using Visual Studio, add a new class called SimpleEmailSender to the Tutorials library and write the following 'OnConstruct' event for the SimpleEmailSender class:

```
public class SimpleEmailSender : Events
{
    protected override void OnConstruct ( Container group )
    {
```

```
            base.OnCreate( group );
            group. Feld("dfFrom").Title = "From";
            group.Field("dfFrom").Editable =  true;

            group.Field("dfTo").Title = "To";
            group.Field("dfTo").Editable =  false;
            group.Field("dfTo").TextTransparent =  true;
            group.Field("dfTo").StrValue = "Target@email";
            group.Field("dfTo").Border =  LineStyle.None;

            group.Field("dfSubject").Title = "Subject";
            group.Field("dfSubject").Editable =  true;
            group.Field("dfSubject").Dimension.Width = 200;

            group.Field("dfMessage").Title = "Message";
            group.Field("dfMessage").Height = 5;
            group.Field("dfMessage").Editable =  true;
            group.Field("dfMessage").Dimension.Width = 200;

            group.PageLink("pbSend").StrValue = "Send";
      }
}
```

The new class SimpleEmailSender now needs to be assigned to the group placeholder created earlier (also called SimpleEmailSender). In i5 Studio, click on the 'Logic' property tab of the group SimpleEmailSender, select Tutorials as the Library Name, click Next and then assign SimpleEmailSender as the Class Name.

At present, of course, the dfTo field on the email form is read-only, which considerably limits the flexibility of the component. Also, lacking code for an OnSubmit event, we are not able to use the form actually to send email anywhere either.

The following code, added to the SimpleEmailSender class, will allow us to send email when the pbSend button is pressed:

```
      protected override void OnSubmit( Container group )
      {
            base.OnSubmit( group );
            if (group.LinkPressed == "pbSend")
            {           // send email
                  using ( Mail myMail = new Mail())
                  {
                        myMail.Host = "Invalid host"; //ie set mail host address
                        myMail.From = group.Field("dfFrom").StrValue;
                        myMail.To = group.Field("dfTo").StrValue;
                        myMail.Subject = group.Field("dfSubject").StrValue;
                        myMail.Message = group.Field("dfMessage").StrValue;
                        myMail.Send;
```

```
                        }
                }
        }
```

To be able to use the email form we now need to allow configuration of the target email address and the email host (both are currently invalid). This is done by exposing the properties of the component in i5 Studio. We will call the properties that we wish to expose 'Email To' and 'Email Host IP'. The following code added to the SimpleEmailSender class will define these exposed properties:

```
private String email_to;
private String email_host_ip;
public String Email_To
{
        get { return this.email_to; }
        set { this.email_to = value; }
}
public String Email_Host_IP
{
        get { return this.email_host_ip; }
        set { this.email_host_ip = value; }
}
```

In the OnConstruct event, modify the definition of the field dfTo so that its value is set to be that of the property 'Email_To':

```
.....
group.Field("dfTo").Title = "To";
group.Field("dfTo").Editable =  false;
group.Field("dfTo").TextTransparent =  true;
group.Field("dfTo").StrValue = Email_To;
group.Field("dfTo").Border = LineStyle.None;
.....
```

In the OnSubmit event set the mail host of the Mail object to the property "Email_Host_IP".

```
        .....
if (group.LinkPressed == "pbSend")
{ // send email
  using (Mail myMail = new Mail())
  {
    myMail.Host = Email_Host_IP; //ie set mail host address                    .....
```

Rebuild the myComponents.dll back into the i5 Lib directory. The Properties tab of the group SimpleEmailForm in i5 Studio will now display the new properties 'Email To' and 'Email Host IP'.

This final fragment of code adds the class configuration function to the SimpleEmailSender class, allowing the email component to appear on the 'New Component…' list in i5 Studio and to set tooltip text to appear over the component icon on that list:

```
protected override void OnQueryPlugin( PluginAttributes config )
{
        config.Type = PluginType.Group;
        config.Description = "Simple Email Sender";
}
```

An enhanced version of this email component, called 'qsEmailForm', is available as a pre-built i5 "plug-in" component and is part of the i5 Quick Start component toolkit. The source code for the component can be found in the i5 QuickStart directory

You will have noted the absence of any proper error checking in the sample code we have written for our email component. It would be a bit reckless to develop a component for use in real-world systems without any kind of error checking, particularly when working with emails. Error messages generated by the Mail object can be trapped using the Mail.LastError method. There are a number of possible sources of error when working with Mail objects. From the point of view of API, likely errors revolve around links with the mail server you are working with. Likely causes will be: not enough memory to send message, failure to open a TCP/IP connection, failure to contact mail server, connection to mail server being lost, not enough information provided to send message, or the mail server failing to respond. However, in addition to likely problems involved in establishing contact with a mail server, other difficulties may arise once contact has been established. These are more properly considered as SMTP (Simple Mail Transfer Protocol) errors and will typically involve problems such as a mailbox being unavailable, or insufficient storage for delivery of message, bad sequences of commands and so on. For further information on SMTP errors, we recommend that you refer to the original RFC establishing the protocol on the Internet Engineering Task Force website. See: http://www.ietf.org/rfc/rfc0821.txt.

About plug-in component class properties

Data types

A property can be declared as any numeric, string, Boolean or enumeration data type. To declare a property of type String:

```
private String enter_some_text;
public Enter_Some_Text
{
        get { return enter_some_text; }
        set { enter_some_text = value; }
}
```

To declare a Numeric property:

```
private Double enter_a_number;
public Enter_A_Number
{
        get { return enter_a_number; }
        set { enter_a_number = value; }
}
```

Both string and number properties appear on the Properties tab of the component in i5 Studio as text input boxes. The property name is used as the title for the input box.

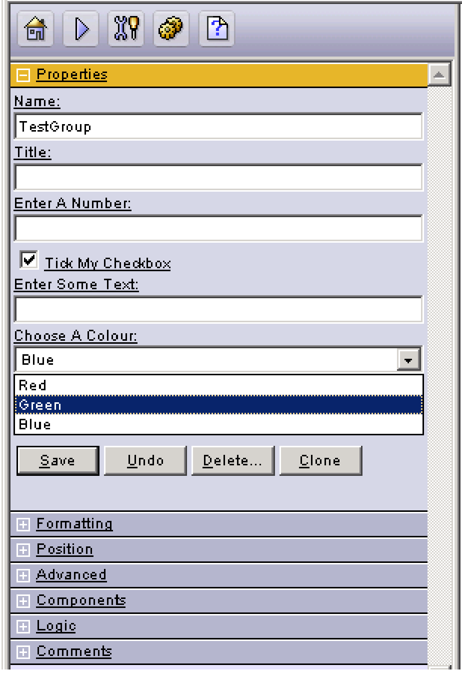
To declare a property of type Boolean, which will appear in i5 Studio as a checkbox, allowing a property to be set to *True* or *False*:

```
private Bool tick_my_checkbox;
public Tick_My_Checkbox
{
        get { return tick_my_checkbox; }
        set ( tick_my_checkbox = value; }
}
```

Enumeration data types allow the user to set a property value by choosing the value from a drop-down list. The list contents must be declared as an enumeration data type in the business logic library. A property which displays the list must then be declared as being of that enumeration data type.

To declare a property that allows the user to choose a colour red, green or blue, declare a 'Colour' enumeration and a public property of that type

```
public enum Color
{
        Red,
        Green,
        Blue,
}
public Color Choose_A_Colour;
```



Default Values

To create default values for the properties, the properties must be set to the desired values by the constructor of the plug-in business logic class. Depending on the programming language used this constructor will be named differently. In C# the constructor is given the same name as the class itself.

For example, to default the property values declared above, you will need to add a constructor to your business logic class

```
public MyClass()
{
        enter_some_text = "Default text";
        enter_a_number = "12345";
        tick_my_checkbox = true;
        choose_a_colour = Color.Blue;
}
```

These default values are only assigned to the plug-in components properties when a new instance of the component is created in the designer.

Concluding Remarks

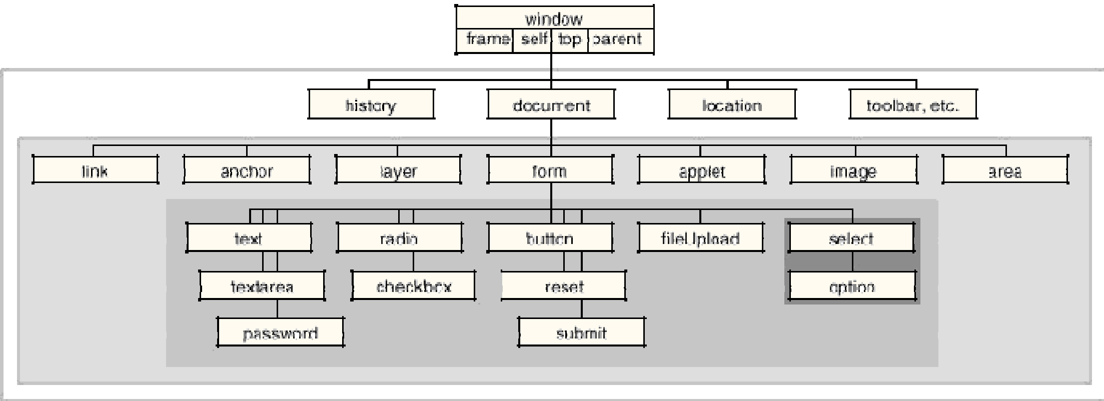
That's it for our quick overview of the use of plugins in i5 3.5. As our overview should have suggested to you, plugins provide a very neat way of putting functionality that you may want to use across applications or across sections of an application into components that can then be accessed by a page designer. In a sense this facilitates the division of labour in the development of applications, as it allows the more technically literate programmer to write sophisticated code for re-usable functionality that can then be made available to developers who simply want to utilise the components, rather than have to worry about what goes on under the bonnet. A number of plugins (and their source code) are supplied as standard with i5. We suggest you take

a look at the code for these plugins if you want to get a more detailed idea of the kinds of things that you can do with plugin components.

6. JavaScript Issues

If you are reading this, it is probably safe to assume that you have done at least some more than basic web development. If this is true, then you have almost certainly worked with JavaScript. We will assume here that you have at least *heard* of JavaScript, even if you've never actually used it. This chapter is about the role of JavaScript in i5 books. In many respects, as a client-side scripting tool, JavaScript is the sticking plaster of web development – it allows you to validate form data, create cookies, check browsers, write HTML on the fly, - in short, to perform a great many small programming tasks which it would be pointless to code into business logic sitting on your server. Whilst JavaScript is not a full-on programming language (it is *interpreted*, meaning that it does not have to be compiled down into machine code before being run, unlike, say, C++), it has a syntax which looks very much like a full-on programming language, with a series of syntactic constructs which operate in much the same way as its more sophisticated compileable brethren.

JavaScript is designed as a series of objects which browsers are able to interpret and which provide ready access to objects written in HTML such as forms, frames, tables, page links and so on. Because i5 operates by sending HTML to a browser, JavaScript works equally well with i5 objects, thus providing a quick, easy and powerful way to access and work with objects in i5 applications and their properties. The diagram below provides a schematic representation of the hierarchy of JavaScript objects. It is worth noting though that not all browsers interpret JavaScript in the same way, which can cause problems for the unwary.



Attempts have been made to resolve this issue in recent years, and the introduction of the W3C *HTML DOM* (Document Object Model), should, in time, standardise the ways that browsers allow access to objects written in HTML such as forms, frames, tables, page links and so on. The HTML DOM basically provides a way of representing web pages schematically, as a tree structure of elements and text nested inside other elements and, theoretically at least, maps onto JavaScript objects neatly and cleanly. It is well worth checking the W3C website for information, particularly for information on the way that major browsers, such as IE and Firefox handle JavaScript. See http://www.w3schools.com/js/js_obj_htmldom.asp for further information

Using JavaScript in i5 to do data validation

Let's work through an example to see how JavaScript works with i5 in practice. Several JavaScript objects are crucial from the point of view of accessing data on an i5 page. An i5 page per se is considered to be a *form* in a *document*. Forms are referenced by name, as are elements on a form, such as datafields. Elements on a form have a value. So, to retrieve the input on an i5 datafield named dfSurname on a detail page named Detail , you would use the following syntax:

```
Document.Detail.dfSurname.value
```

Let's try a simple example to get us going. The following code fragment is used to validate the data input for the height of a patient whose details are being taken at a clinic:

```
if (document.Detail.dfPatientHeight.value – 10 >= 0)
{
        alert('Patient height must be less than 10 m!');
        return false;
}
```

Here we have simply retrieved the value of the dfPatientHeight field, run a check on it and warned the user that no human can be ten or more metres tall.

The 'linkPressed' syntax is useful when validating data on a save. So, in the example below, we warn the user that a date has not been entered when they click on the button pbSave:

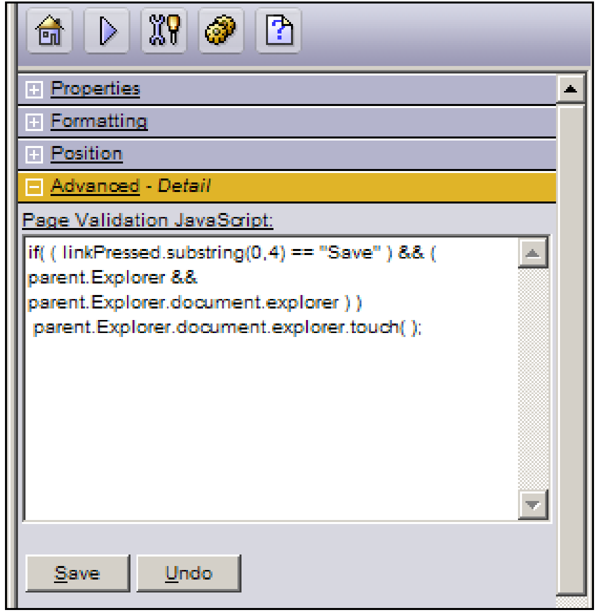
```
if ((linkPressed == "pbSave") &&
(document.Detail.MyDateLookup_d_0.value == ''))
{
        Alert('Please enter a date');
        return false;
}
```

However, this simple piece of validation script will not be invoked unless we also tell the page link that we have referenced – pbSave – to invoke page validation script. You can do this simply by checking the 'Invoke Page Validation' property on the Properties tab of the page link in question.

Note the _d_0 suffix added to the datelookup object. This is required because the datelookup object is a group component and we require some way of referencing individual component elements of a group.

Where do I put it?

Typically, JavaScript for data validation in i5 books is added to pages via the Page Validation JavaScript field on the Advanced tab of the page in question. If the JavaScript which you add to your page returns 'false', as in the example above, changes are not saved to the database. If your script returns 'true', default form validation is bypassed. Specifying no return value allows edits to be saved normally.



The Page Validation JavaScript property of a page is not the only place you can locate your code. In the examples we are going to explore below, we use a standard i5 text field to store our JavaScript in. The other possibility, which is particularly useful if for some reason you have written a number of functions, say, is to save the bulk of your JavaScript as a completely separate file altogether. In this instance, you would save the file with the file extension .js and store it in the root directory of your webserver (in IIS this is wwwroot). However, you do then need to reference the file on the page which calls particular functions. You would do this by adding an invisible Text object to your page, of type HTML, with a line of HTML in it similar to the following:

```
<script language=”javascript” src=”/mySite.js”></script>
```

Now, any JavaScript function which you call on this page (or, indeed, on any other which references this script) can be collected and stored in one script in a separate location. This makes managing your code easier.

Exploiting Properties, Methods and Event Handlers

All JavaScript objects have series of *properties, methods* and *event handlers* associated with them. In the code fragments above, 'value' is a property of a *text* object and 'alert' is a method of the root (browser) window object within which i5 books appear. Event handlers listen out for particular user actions, such as mousing over a datafield or clicking on a button. The ability to access properties, call methods and handle events using JavaScript objects from within i5 pages is enormously valuable and makes it possible to do far more than simply validate data.

Let’s have a look then at using JavaScript to help us print a properly formatted report based on data populating an i5 page. To do this we actually require two i5 books. In BookMain we will have a section MyReports with a detail page MyFirstReport populated with data from some database table. We use the data from this table to populate a child table MyQueryResults which sits on the MyFirstReport detail page. A second book, BookReports connects to the same database, and sits in the same directory as BookMain. It also has a section MyReports and a detail page MyFirstReport, with a table MyQueryResults. Unlike the MyFirst Report page in BookMain, BookReport.MyReports.MyFirstReport provides no filtering, because all we want it to do is to display the result set which we have created through filtering in our main book. In

order to do this, however, the child tables in both books require a hidden datafield to set a bind variable for use in the 'Where' clause of the SQL statement used to populate the table. Let's call this field dfMyFilter on the child table in both books. In order to set a value for dfMyFilter, we will add an appropriately populated listbox to the page in BookMain. Let's call that listbox lbMyFilter. In the SQL statement for the child table in BookMain, we must therefore set our 'Where' clause to filter the appropriate column in the database table by the value retrieved from the listbox (we use the bind variable syntax :lbMyFilter to do this). In the SQL statement for the child table in BookReport, we set the 'Where' clause equal to the bind variable :dfMyFilter.

Leaving aside the details of how we format the pages and so on, let's look at the JavaScript we need. Rather than use business logic, or the dedicated JavaScript object on the i5 page, we will add what we need by creating a button in HTML. This will actually serve as our 'print' button.

To create a 'print' button, add a text object of type 'HTML' to the report page in your main book, and set the text equal to the following:

```
<br>
<button type="button" name="pbPrint"  onClick="window.open ('/scripts/i5.exe?book=
  BookReports&page=MyReports.MyFirstReport&dfMyFilter=' +
  document.MyFirstReport.lbFilter.value,'','toolbar=no,location=no,directories=no,
  status=no, menubar=no, scrollbars=yes, resizable= no, width=670, height=600,
  left=50, top=50');  return false;" accesskey="P">
Print Report
</button>
```

There are several things to note about this code. In the first place, we have inserted the JavaScript into the HTML for the button object. In the second place, this JavaScript uses the onClick() event handler to open up the correct page in our book of reports. It also sets the filter value for the 'Where' clause of the SQL which selects the data to populate the child table on the report page. It takes this value from the lbFilter object in the main book. It also sets a whole host of other properties – primarily stripping the browser window down so that when it prints nothing but the information it has extracted from the i5 book will be rendered.

However, note that this JavaScript will not itself be sufficient to print out our report. It only formats the browser window appropriately. To print the report, we need to make use of Cascading Style Sheets.

Invoking JavaScript from a CSS file

If you've done much work in i5, you will doubtless have come across Cascading Style Sheets. In fact, whenever you create a page in i5, the formatting tab of the page object invites you to specify an HTML template to provide a uniform styling for your pages. These templates are based on the CSS syntax. By default, i5 comes with a selection of templates, which range from the cheesy to the really cheesy (they are located in the 'Include' subdirectory of the main 'Dataline\i5' installation directory). Conveniently for us, Cascading Style Sheets not only let us define aspects of the rendering of an i5 page such as font, background colour and so on, they also allow us to invoke JavaScript.

Create an .html file as follows:

```
<html>
<head><title>Print Report</title></head>
<style type="text/css">
```

```
<!--
.content {
		font-family: Arial, Helvetica, sans-serif;
		font-size: 10px;
		color: #666666;
}
-->
</style>
<body marginheight="0" marginwidth="0" leftmargin="0"
topmargin="0" bgcolor="#ffffff" onLoad="window.print();">
<screen>
</body>
</html>
```

Note the use of the text/css value for the style type attribute. There are a variety of attributes set here in the body tag – obviously these will vary according to how you want your pages to render. But the key attribute to specify here concerns the onLoad event, and the proprietary i5 <screen> tag also specified within the <body> tag. Setting onLoad equal to window.print() calls up a print dialog, whilst the <screen> tag is used by i5 to determine where to insert the specified HTML. Save the file, as something like 'PrintReport.html', in the 'Include' subdirectory of your i5 installation, and then set the 'HTML_Template' property of the detail page in your reports book accordingly. If you have followed everything correctly to this point, you should now find that whenever you click the 'Print' button in your main book, a secondary browser window will pop up, followed a split second later by a 'Print' dialog box.

Using JavaScript for timeout problems

We can also use JavaScript for other things. Let's take a look at one last example. This time we are going to use JavaScript to solve an engineering problem. We have a multi-frame book and we are using a standard i5 login object to authenticate users. The problem we have is a simple. For security reasons, when a timeout occurs – perhaps because a user has left his or her machine unattended for too long – it would be prudent to reset the pages in our book. Displaying potentially sensitive data which the user might have been working with and then leaving a machine unattended is rarely a good idea. In an i5 book without frames, this is not an issue because there is only one page to reset when the timeout occurs. However, if we are using frames, we do need to be sure that we can reset the pages which are displayed in frames other than that which contains the primary login object. The difficulty is really quite simple. This resetting process does not happen automatically. Fortunately, a judicious use of JavaScript will allow us to resolve our problem. The trick is basically to check the connection handle to the database and in case of a timeout, simply reload the book as a whole.

Let's just explore this in a little more detail. For a book with a frameset with three frames we need login pages for each frame, one assigned to each frame, as below:

- MyTestBook
  - Frame Layout
    - FrameLeft
      - FrameTop
      - FrameBottom
        - new *FrameBottom* component..
      - new *FrameLeft* component..
    - FrameRight
      - new *FrameRight* component..
    - new *Frame Layout* component..
  - Login
    - Login
      - u
      - p
      - pbLogin
      - new *Login* component..
    - LoginBtmLeft
      - new *LoginBtmLeft* component..
    - LoginRight
      - new *LoginRight* component..
    - new *Login* component..
  - new section...

If we want to add a series of pages – one for each frame – for the user to go to on login, we would create a default section for the book, containing three pages, one assigned to each frame. Let's call such a section something like 'HomeSection'. Remember that the default section is specified on the Properties tab of the book object in the Explorer view of the Page Designer. The pages defined in HomeSection will now be the pages to which the user is taken *by default* on login. The crucial point, and this is what necessitates adding some JavaScript to our book, is that when a timeout occurs, *only* the frame containing the main login page will be reset, which is not the behaviour that we want.

To resolve this problem, we need to add some JavaScript to the main Login page. To do this, add a Text object, specified as HTML, rather than plain text, to the main Login page and cut and paste the following JavaScript code into it:

```
<script language="javascript">
if ( c != '' && typeof(design) == "undefined" )
{
alert ( 'Your connection has timed-out' );
window.parent.location = 'i5.exe?book=' + book;
}
</script>
```

This JavaScript does several things. It checks to see if there is a connection handle to the database. It also exploits the JavaScript operator typeof to check whether the user is in i5 design mode or not (this is for convenience really: it ensures that the subsequent resetting of the book only occurs in runtime mode). If the condition evaluates positively, it then warns the user that a timeout has occurred and, using the JavaScript Location object, it resets the CGI parameter required by the i5 runtime engine so as to reload the book. This will then reset the pages in all three frames back to initial login, solving our engineering problem elegantly and efficiently.

The ubiquitous <div> tag

Unless you have been writing code in a time capsule recently, you cannot fail to have encountered AJAX (Asynchronous JavaScript and XML). Dataline have developed a plugin AJAX component, which you can download from the website, and for which there is a

whitepaper detailing its use. One of the enormous advantages of AJAX is that it allows you to update targetted areas of a webpage independently of the page as a whole (that is a crude way of explaining what the term 'asynchronous' means). The implications of this possibility are extensive and we do not have space to go through them all here. However consider the situation where you have a webpage that is populated using a complicated SQL query using multiple joins on multiple tables in a database. Such a page may take some time to load. You may have a component on that page which you want a user to update before accomplishing some more complicated database transaction for the page as a whole. In these circumstances, it would be 'uneconomical' to force a refresh of the page as a whole just for the purposes of allowing the component update to take place. If you can update different parts of the page asynchronously, you can avoid this problem, and thus you can update your component without having to entail all the resource overhead of processing the page as whole. Now, rather than go through that scenario here, we will just wrap up our discussion of JavaScript, by demonstrating a very simple use of the <div> tag that AJAx relies on, in an i5 book.

Let's imagine that you want to have a section on an i5 page that is only displayed when the user clicks on an 'expander' button. We can exploit <div> tags and the CSS 'display' property to do this.

Create an i5 page and add a group to it – call it say, 'grpDisplay'. Put a text object with some or other message to display in the group. Above the group in the Designer, you now need to create two Text objects. Call one 'txtDivStart' and the other 'txtDivStop'. Set the data type of both Text objects to 'HTML' and align txtDivStart to the left margin and txtDivStop to below the previous item. Re-position the grpDisplay object so that it is aligned below txtDivStop (obviously you will need to use flow-based layout on this page). Add the following script to the Text property of txtDivStart:

```
<div id="HiddenDataHeader"
    style="width:510px;
    height:6px;
    background-color:#F2F2F2;
    padding:1px;cursor:pointer;"
    onClick="openclose('HiddenData', 'fr_ecarrow')"
    title="Click to view/update my hidden information"
    <span style="float:left; font-weight:bold;">View/Update Hidden Information
    </span>
    <img src="/i5/Expand.gif" border="0" align="absmiddle" style="float:left;" id="fr_ecarrow">
</div>
```

And the following script to the Text property of txtDivStop:

```
<div id="HiddenData" style="width:250px; border:1px #c6c6c6 solid; background-color:#F2F2F2; border-
top:none; display:none;">
<img src="/Dataline/Images/blank.gif" width="1" height="1">
```

Finally, add another Text object to the page – call it something like 'txtJavascript' and set its DataType to 'HTML'. Set its Text property to read as follows:

```
<script>
        function openclose(divid, imageid)
        {
                var sp = document.getElementById(divid);
                var ec = document.getElementById(imageid);
```

```
                if(sp.style.display == 'none')
                {
                        sp.style.display = 'block';
                        ec.src = '/i5/Contract.gif'
                }
                else
                {
                        sp.style.display = 'none';
                        ec.src = '/i5/Expand.gif'
                }
        }
</script>
```

Note that we haven't actually provided you with the Contract, Expand and blank GIF images that we reference in this script – you will need to create these yourself and locate them in the appropriate directories. Anyway, provided that you can add these objects with the correct JavaScript in place, you should find that you now have an expandable section on your page, that displays your hidden message when you click on your expander icon.

This is a fairly rudimentary example of the use of the <div> tag and JavaScript to add functionality to an i5 page. There are, of course, a much broader range of options open to you if you are willing to experiment. It is worth keeping an eye on the i5 section of the Dataline online forum, which can be accessed at http://www.dataline.co.uk/forum/ for discussions and tips regarding the more advanced use of i5.

Concluding Remarks

That's about it for our exploration of JavaScript here. As mentioned earlier, it functions very much like a glue for web applications. We have barely even scratched the surface of how it can be employed in i5. The examples developed here are really only supposed to give you a taste of what you can achieve by the judicious application of client-side scripting to your i5 website. Obviously, we do not recommend that you use JavaScript for sophisticated or heavy-duty data processing – database servers and business logic are optimal for such purposes. However, as you have seen, for accomplishing data validation and solving minor engineering problems, JavaScript is an excellent weapon to have in your arsenal, and the additional flexibility that AJAX offers you by way of asynchronous updating of parts of a page provides an excellent extension to the functionality that you can provide on an i5 page.

This is also the end of the *Developer's Guide*. As with the JavaScript we have explored in this chapter, we've only really scratched the surface of what i5 can do in terms of building data-driven web applications. We hope that the discussion and the examples have been helpful. More technical information about the classes available to you and the methods that they implement is offered in the i5 3.5 .Net API reference. We also recommend that you visit the Dataline Forum for a more interactive resource supporting your development projects. The forum is accessible online at http://www.dataline.co.uk/forum/YaBB.pl