



# T320 E-business technologies: foundations and practice

## Block 3 Part 1 Activity 5: Implementing a simple web service

Prepared for the course team by Neil Simpkins

---

Introduction	1
Components of a web service	2
Implementation approaches	2
Eclipse for Block 3	3
Steps to a first web service	3
Create a project	4
Writing the service logic	8
Generating a web service and client	12
Testing the web service	17
Using the client	17
Monitoring web service messages	19
Service descriptions	19
Web service endpoint	19
Axis and WSDL	21
Summary	24
Appendix 1: WSDL generated for 'Hello' web service	24

---

## Introduction

In this activity you will see how to use Eclipse to create and test a very simple web service. Before I look at the service itself, however, I shall provide some background information about the different approaches that can be used to develop a web service.

## Components of a web service

From an implementation point of view, a web service typically has two distinct components:

- Some 'logic', which actually carries out the real work required to provide the service functions, whatever they might be. For example, a simple service to calculate VAT might include at its heart a piece of code that receives a monetary value in sterling, such as £10, multiplies that value by 17.5% to calculate the VAT (£1.75) and returns the new total value including VAT (£11.75).
- An interface, which is implemented in XML. This describes how to put a request to the service and what the expected response or result may consist of.

In addition, in order to actually make use of any web service we implement, we need also to implement a client that constructs our request (based on the XML interface) and displays the result returned by the web service.

This view of a web service should help to explain the different approaches to producing a web service.

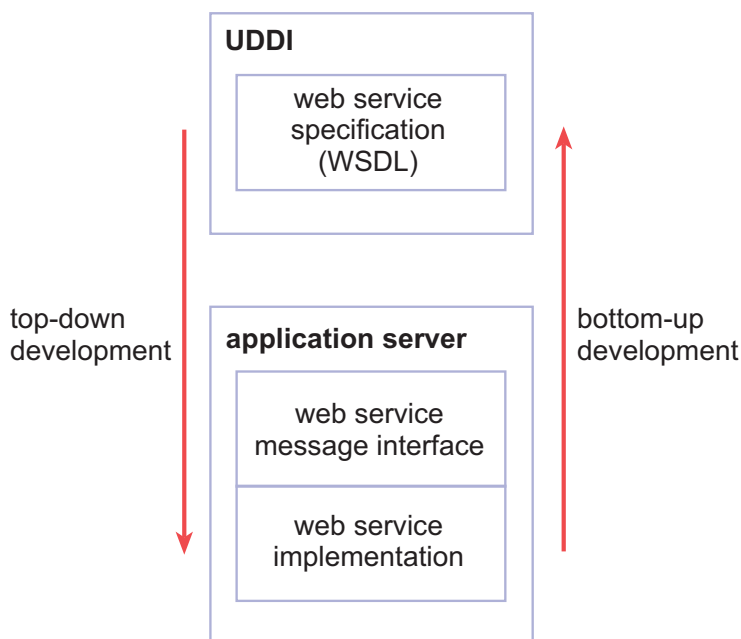
## Implementation approaches

There are two basic approaches that are used to create web services from scratch.

The first approach is termed 'bottom-up'. This is where the code that implements the operations performed by the service is written first. Then the XML description (WSDL) of the service is produced and published in UDDI. The service code is placed inside a 'container' that provides the required interface for messaging (SOAP or something similar).

The second approach is to produce the XML description of the service before it is implemented. So the service is fully described in terms of what it does, how it can be called and what result(s) it returns, but is not actually implemented. This (WSDL) specification is then used as a guide to writing the code that implements the service. This approach is termed 'top-down'.

These two approaches are illustrated in Figure 1.



**Figure 1** Bottom-up and top-down web service development

## Eclipse for Block 3

You should have installed the software required for the practical activities in Block 3 when you were working on Block 2. If you have any problems with the installation process then please go to the course forums for guidance or contact your tutor.

Block 3 uses Eclipse, Java, Tomcat and the following plug-ins (extensions) to Eclipse. Further information about these plug-ins is available through the 'Library resources' section of the course web site if you are interested.

### Axis2 plug-ins

Two Axis2 plug-ins for Eclipse are used in this block. In fact you will use only one of these explicitly: the 'Service Archive Generator Wizard'. For now you don't need to worry about this, as it will be introduced more fully later.

The other Axis2 plug-in is grandly called the 'Code Generator Wizard'. This plug-in can be used to support either top-down or bottom-up development. It will be used in this activity, but the use will be hidden from you. If you wish, however, you can go on to use this plug-in directly yourself, either bottom-up or top-down.

### WTP plug-ins

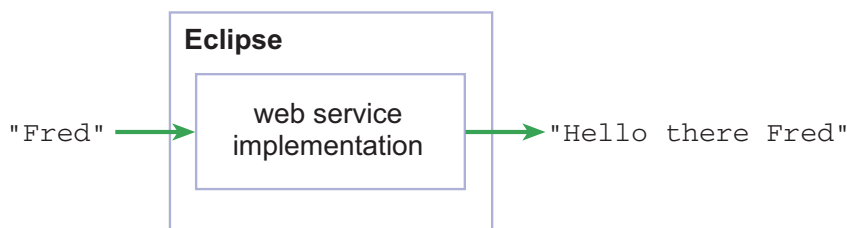
The Web Tools Platform (WTP) is an Eclipse project that extends Eclipse with a range of tools for developing, running and testing web services. In Block 2 you used the XML editor provided by the WTP, while in this block you will use much of the remaining functionality.

### soapUI plug-in

This plug-in has a range of functions, several of which are similar to the WTP's functionality. It is up to you which you prefer to use. However, later in this block you will use the soapUI plug-in to test web services for interoperability, which the WTP doesn't support directly.

## Steps to a first web service

The first web service I shall demonstrate how to write is very straightforward. It will accept a simple piece of text (intended to be a person's name) and return another piece of text as a result ('Hello there' followed by the name sent to the service). This is shown in Figure 2.



**Figure 2** 'Hello' web service outline

Eclipse will help by automating several of the tasks that are involved in the process of developing, testing and deploying the web service.

There are different sequences of steps that can be used in Eclipse to achieve the same end result; here I will try to use the simplest sequence possible. You should already have watched the Block 3 video tutorial, 'Creating a simple web service using Eclipse'. This will have given you an overall idea of the set of steps that you will need to take, as well as some idea of the 'hello there' Java code you will use as the service's logic.

In short, the steps you will take to produce and test your first web service are as follows:

- 1 Create a project in the Eclipse workspace of type 'Dynamic Web Project', which will host your web service.
  - 2 Write the Java code (the 'business logic') that implements your web service functionality.
  - 3 Use Eclipse to automatically generate the components (WSDL etc.) that will transform the Java code into a web service, and then ask Eclipse to run that web service for you.
  - 4 Create another project of type 'Dynamic Web Project', which will host the client application that you will use to access and test the web service.
  - 5 Use Eclipse to automatically generate a set of web pages that function as a client interface to call the web service.
  - 6 Use the web page client to send a request to the web service and observe the web service's response.
- 

### Application server setup

Before you can create and run an application such as a web service, you will need to set up an application server inside Eclipse. To set up the Tomcat server that the T320 software installation has provided, you should follow the guide *Configuring an Application Server in Eclipse*.

---

Once you have set up the application server, you can start working through the instructions given in the following sections.

## Create a project

If Eclipse is not already open, open it by double-clicking on the 'Eclipse for Block 3' shortcut on your desktop. Ensure that you have the Project Explorer open in a Java EE perspective (Figure 3). If you are not currently in this perspective then use Window > Open Perspective and then select 'Java EE', which may be on the menu list or accessible under Other..., depending on your current perspective.

I shall not be going into the details of Java code and the many associated technologies. You might like to know that 'Java EE' is a reference to the Java programming language 'Enterprise Edition'. 'Enterprise' solutions are commonly thought of as reliable and scalable technologies.

The first step is to create a project that will be used to hold all the code and other elements that will make up your web service.

Select File > New > Project..., open up the 'Web' folder and select 'Dynamic Web Project' (Figure 4). Then click 'Next'. This will start the New Dynamic Web Project wizard, which will guide you through the project setup stages.

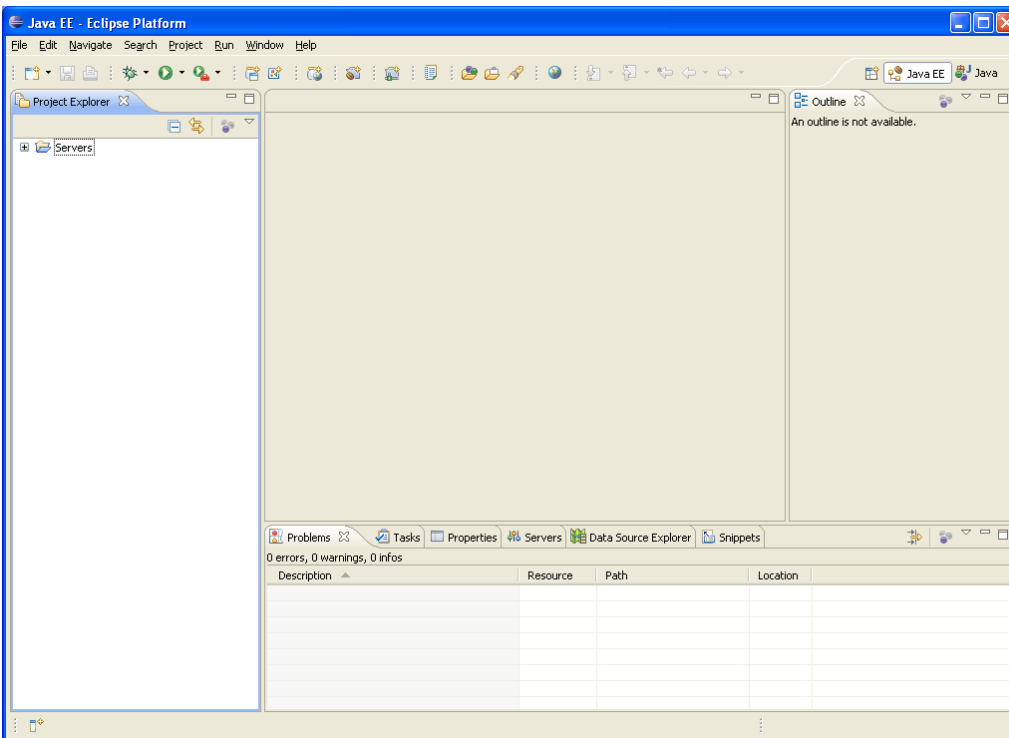


Figure 3 Eclipse Project Explorer in Java EE perspective

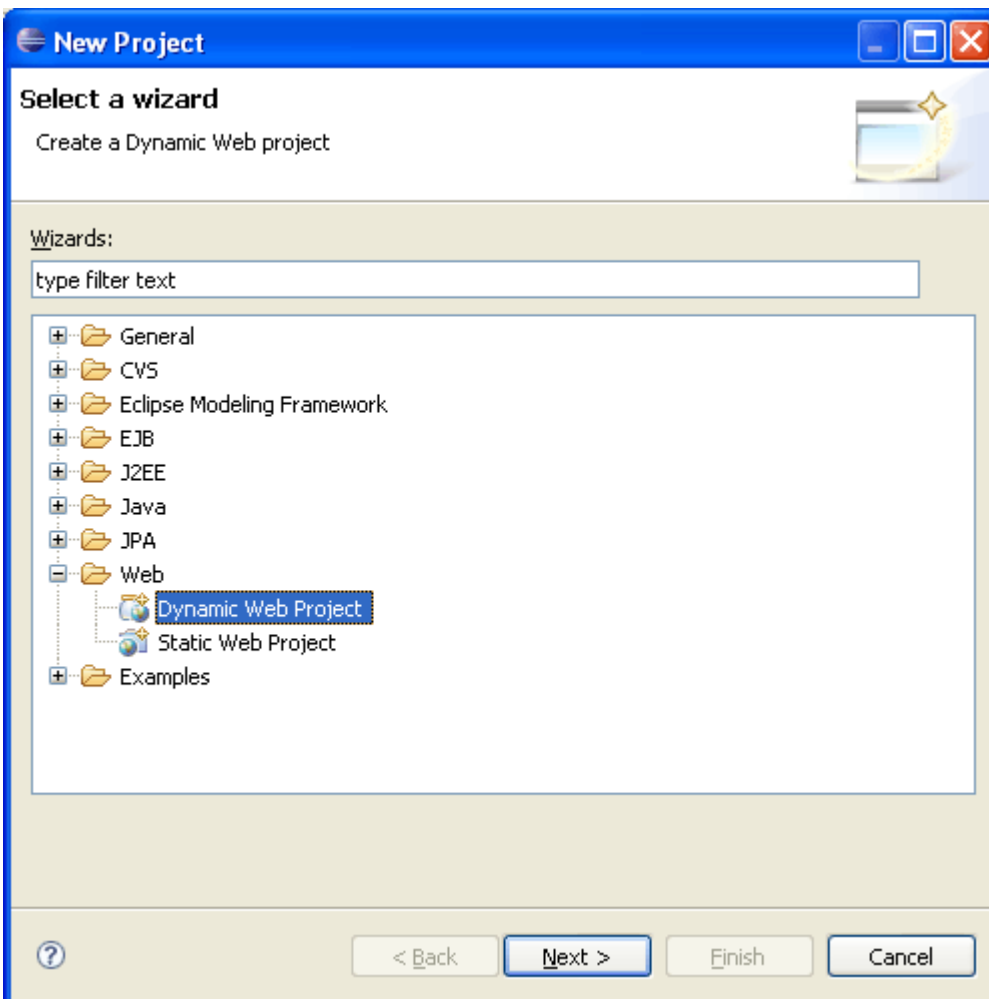
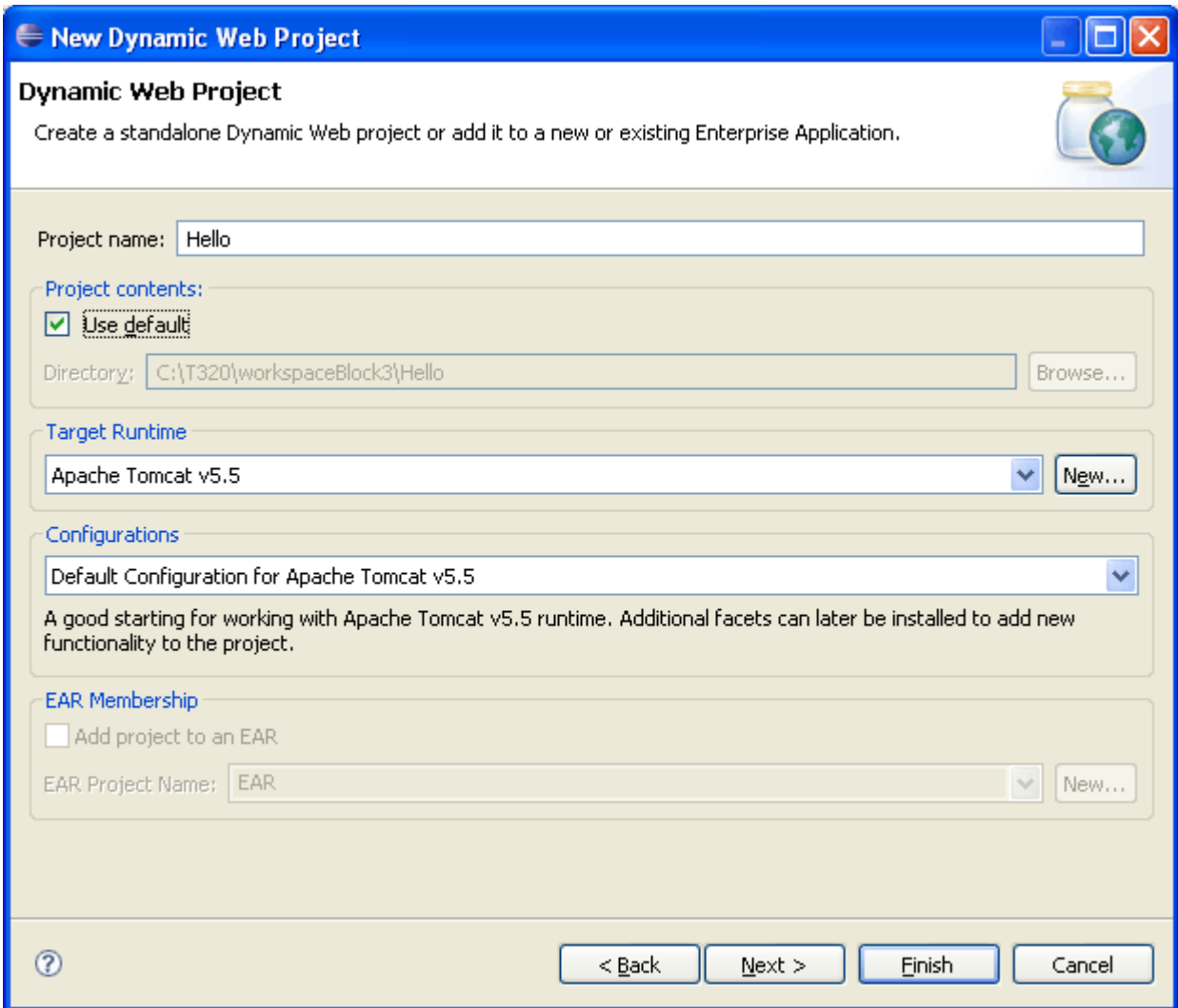


Figure 4 New Project dialogue box

Type a name for your project in the next dialogue box. Here we will call the project 'Hello' (Figure 5). At this point you may either click 'Finish' and have the project created immediately, or click 'Next' to continue and see what other options are available in configuring a project. It's a good idea to become aware of the options available when using an Eclipse wizard, so click 'Next' to continue.



**Figure 5** Dynamic Web Project setup dialogue box

Continue through the wizard's steps as shown in Figure 6 and Figure 7. Note the names of the 'Content' and 'Java Source' directories (Figure 7) and then click 'Finish'.

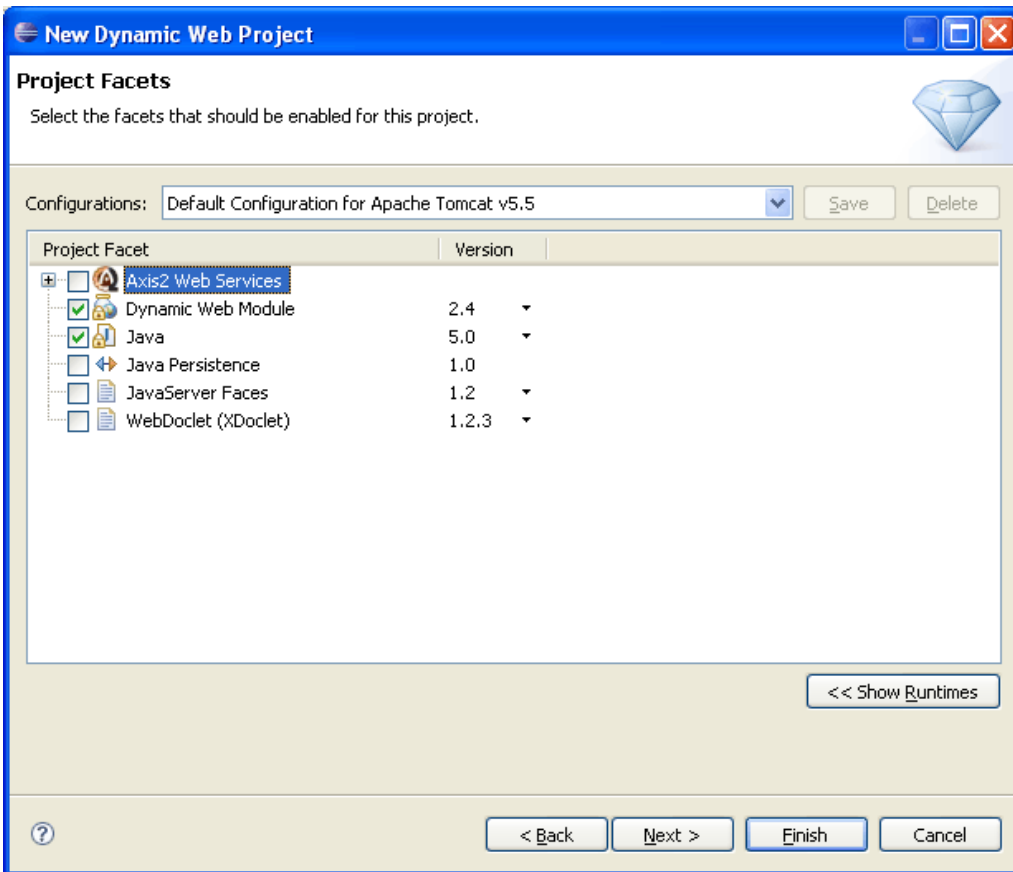


Figure 6 Project Facets choice dialogue box

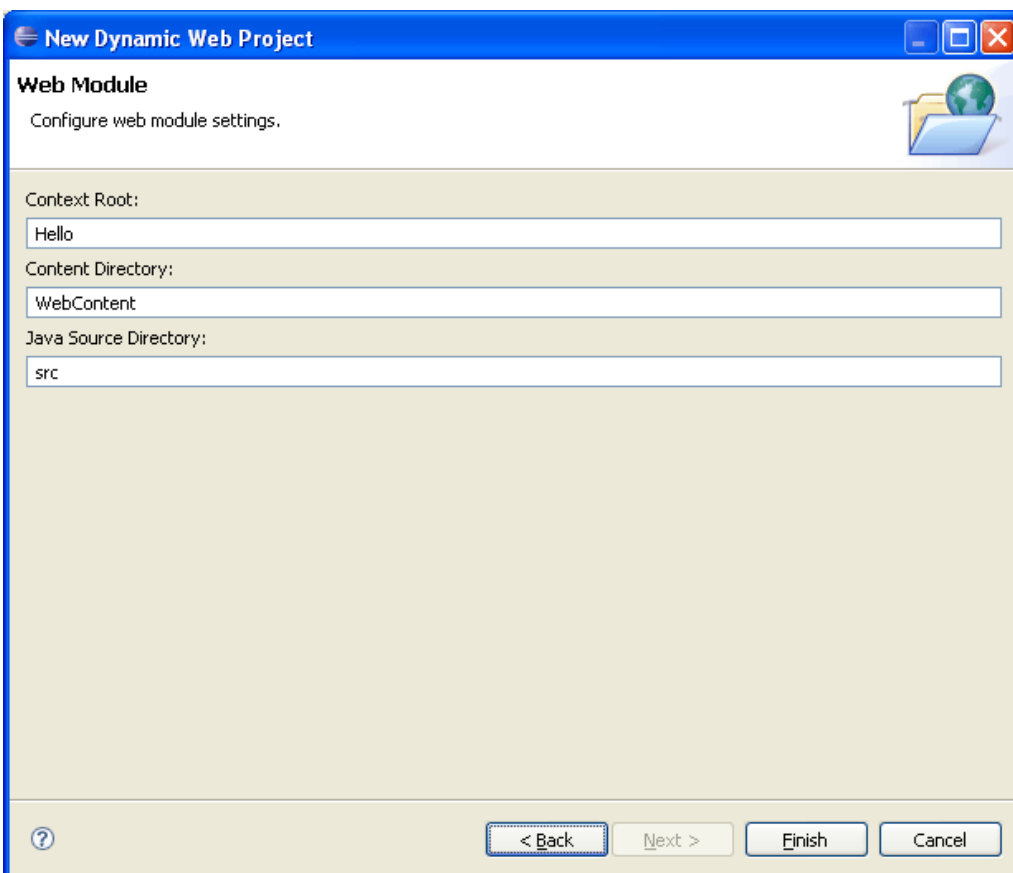
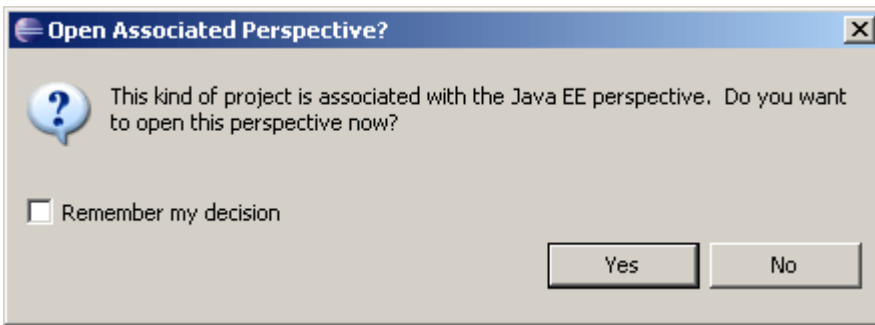


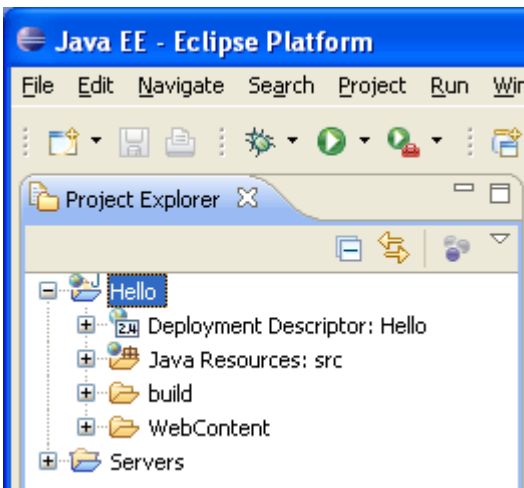
Figure 7 Web Module configuration dialogue box

At this point, if you have not already selected the Java EE perspective then Eclipse will offer to open this for you (Figure 8). If this happens then click 'Yes'.



**Figure 8** Eclipse changes to the appropriate project perspective

Once the wizard has finished you will see that a project called 'Hello' has been created in the Project Explorer view. You can expand the project to view the components (Figure 9).



**Figure 9** Dynamic project created

## Writing the service logic

Now you will create the 'business logic' or processing code that performs the task behind the web service we want to deliver.

The Java code is very simple. Here it is with each line numbered for referencing purposes:

```

1. package uk.ac.open.t320;

2. public class Hello {

3.     public String helloName(String name){
4.         return "Hello there " + name;
5.     }

6. }

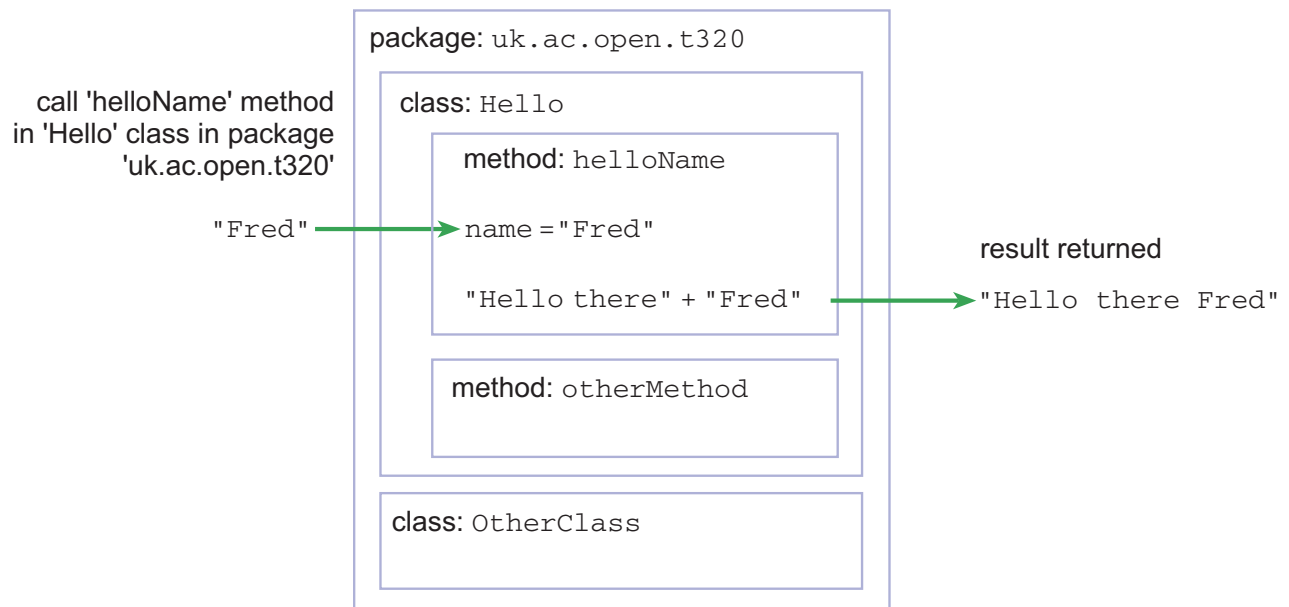
```

I shall not be asking you to write Java code yourself, but you will need to understand what the code used in this block does, at least in high-level terms. Below is a brief line-by-line description of the code, followed by a diagrammatic view. What is important at this stage is that you become familiar with the terminology used.



- Line 1 specifies the Java package name to be `uk.ac.open.t320` (I shall take a brief look at packages very shortly). This 'names' our code as belonging to the T320 course within The Open University, within the 'ac' (academic) domain within the UK. It is not necessary to use a package name at all (as was the case in the video tutorial) but it is recommended by various Java guidelines and Eclipse.
- Line 2 declares a publicly available object or 'class' in Java called 'Hello', which implements the operations that we want our web service to carry out.
- Line 3 declares an operation that can be performed by the 'Hello' class. Operations are called 'methods' in Java. This operation is called 'helloName' and takes an input called 'name', which is a 'String' such as "Fred" or "Nick" or "Sunday afternoon".
- Line 4 is a single Java statement within the 'helloName' method, which produces the method's result by returning a string. This string is created by appending the string that was passed into the method to the string "Hello there".

The operation of the code is depicted in Figure 10, where the 'helloName' method is called and given a string value "Fred". It returns a result "Hello there Fred".



**Figure 10** Calling the 'helloName' method

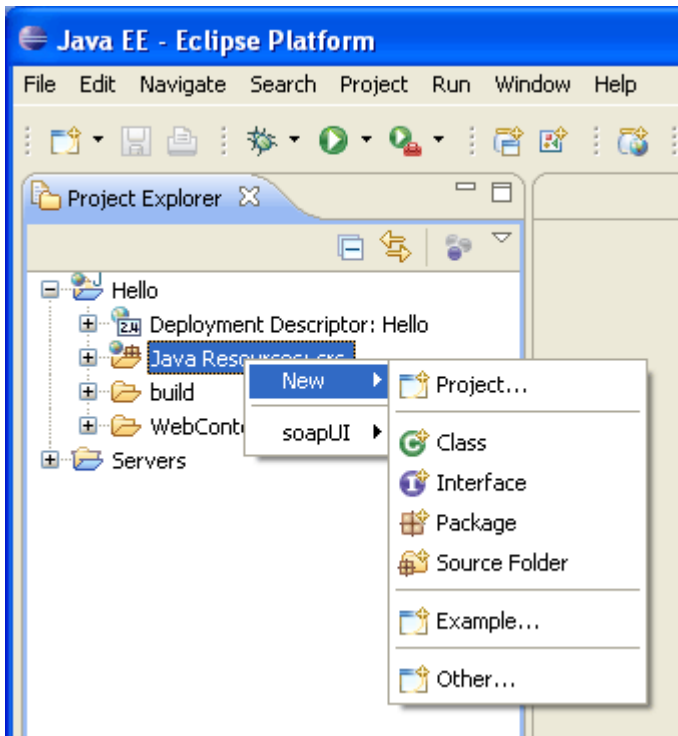
Notice that, as depicted in Figure 10, a package can contain any number of classes and a class can contain any number of methods.

**Warning:** for the purposes of coding in Eclipse, always ensure that method names start with a lower-case letter. Otherwise you will run into problems when attempting to run and test the web service.

As I mentioned above, I have chosen to place the code in a Java package called `uk.ac.open.t320`. This is a matter of style and is not really necessary, but it is good practice to use packages in Java. The package name I have used is based on The Open University's domain name (the OU web site is at <http://www.open.ac.uk>), which is reversed to form the first part of all OU package names. I then added the name of the course within the University, 't320'. This gives us a unique package name that would be very difficult for any other group to legitimately claim as more appropriately theirs – there will only ever be one T320 course within the OU. (If you would like further information, the conventions for naming packages are described in [http://java.sun.com/docs/books/jls/third\\_edition/html/packages.html#7.7](http://java.sun.com/docs/books/jls/third_edition/html/packages.html#7.7).)

Now we need to introduce the code into the 'Hello' project. To do this we could add a package and then add a class to the package. However, a quicker approach is to add a class and specify that it belongs to a package; if the package doesn't exist then Eclipse will create the package so that it can put the class into it.

In the Eclipse Project Explorer, expand the 'Hello' project and right-click on the 'Java Resources: src' component, then select New. The pop-up menu shown in Figure 11 will be displayed.



**Figure 11** New pop-up menu items for Java Resources sources

Select Class and you will be presented with the New Java Class dialogue box (Figure 12).

**It is important that you enter a package name based on your OUCU.** So if your OUCU is 'nks34', you might enter a package name of:

```
uk.ac.open.t320.nks34
```

This will ensure that you have a unique package name, which may well become important to you later in the course.

Then enter the class name in the 'Name' field and click 'Finish'.

When the new class is created, you will see that the package and Java source file ('Hello.java') have been created in the project and that the Java file is opened for editing (Figure 13).

The 'Hello.java' file that is opened for editing contains the package heading you require and a skeleton for the 'Hello' class definition. Now copy the class code given earlier in this document into the editor and save the code using File > Save.

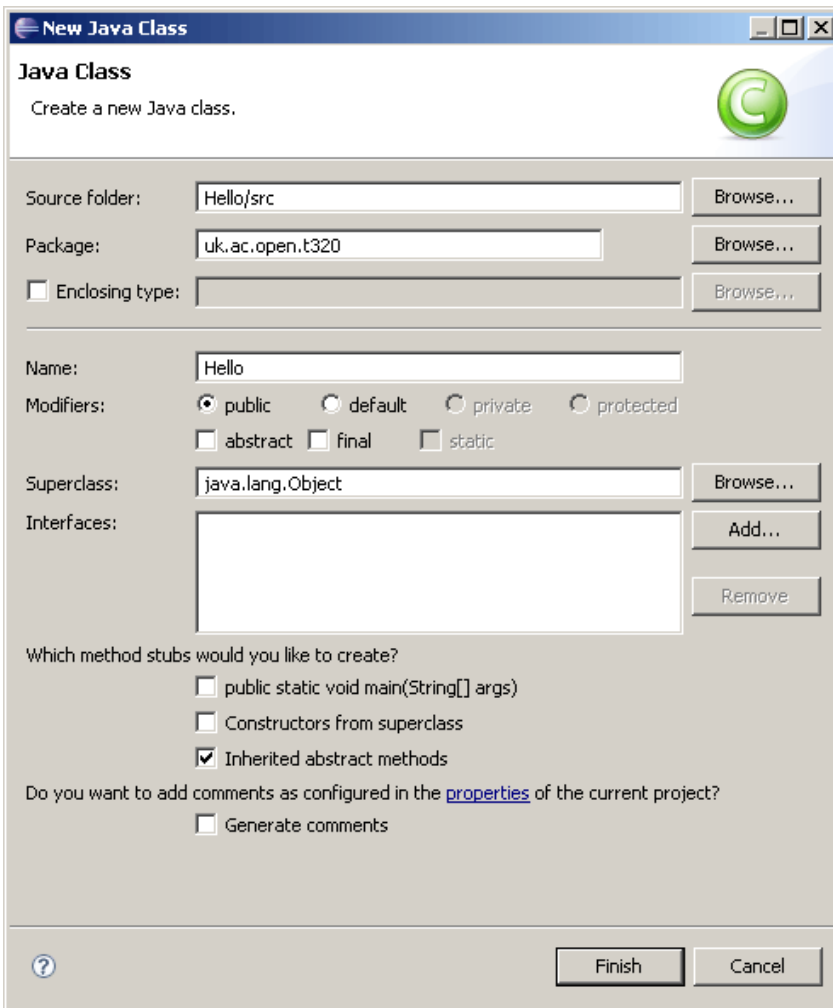


Figure 12 New Java Class dialogue box

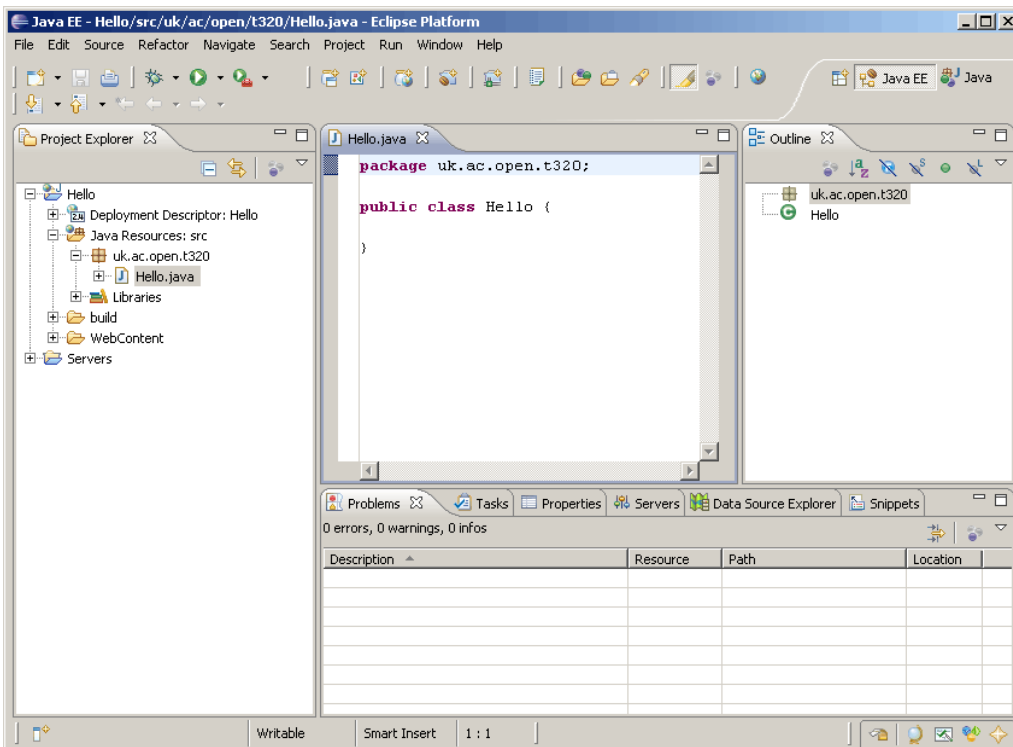


Figure 13 Initial 'Hello' class

If all is well you should find that Eclipse has compiled the source code 'Hello.java' to produce a compiled class file called 'Hello.class'. This file can be found in the project's 'build' directory. The complete project and files are shown in Figure 14.

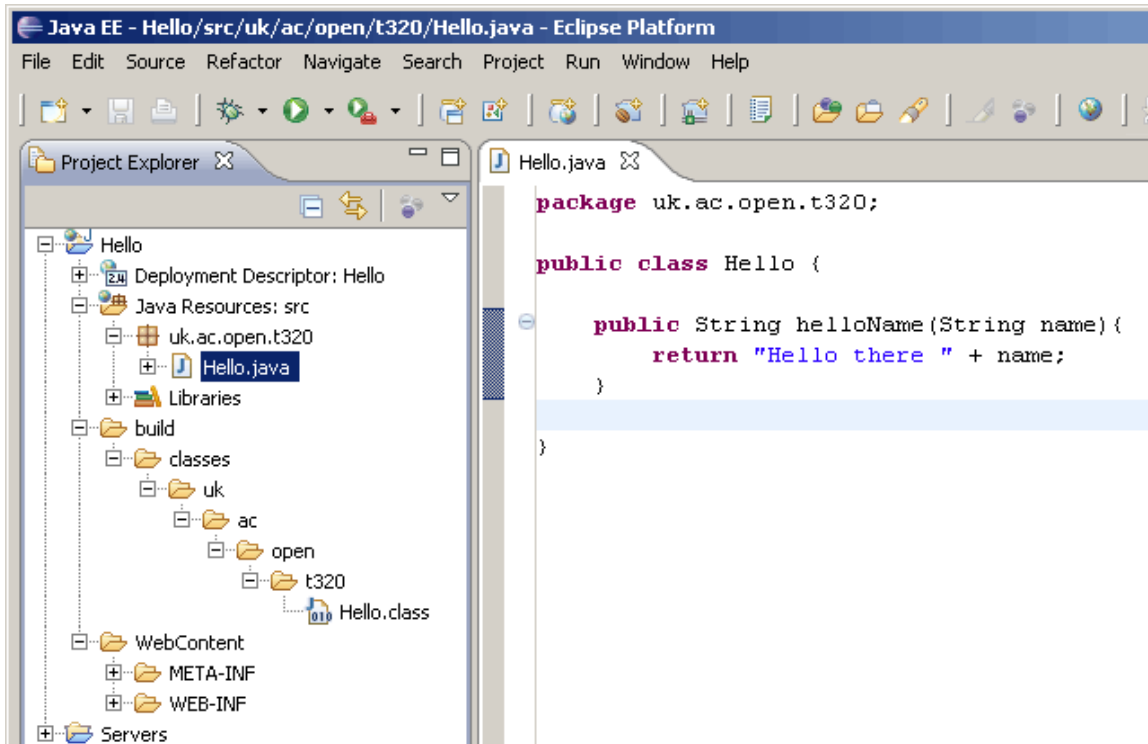


Figure 14 'Hello' project expanded after compilation

## Generating a web service and client

You have now created an empty Dynamic Web Service project and added the code you need to turn this into a web service. Eclipse will do the rest of the work for you by:

- packaging the web service up with an XML interface
- deploying the web service to an application server of your choice (from those you have configured in Eclipse)
- starting the application server if required
- generating a set of web pages that can be used as a simple client to test the web service.

A wizard is provided by Eclipse for these purposes.

Select the 'Hello.java' class in the Project Explorer (highlighted in Figure 14) and go to File > New > Other...; then open the 'Web Services' folder and select 'Web Service' (Figure 15). Then click 'Next'.

The next dialogue box has three sections (Figure 16).

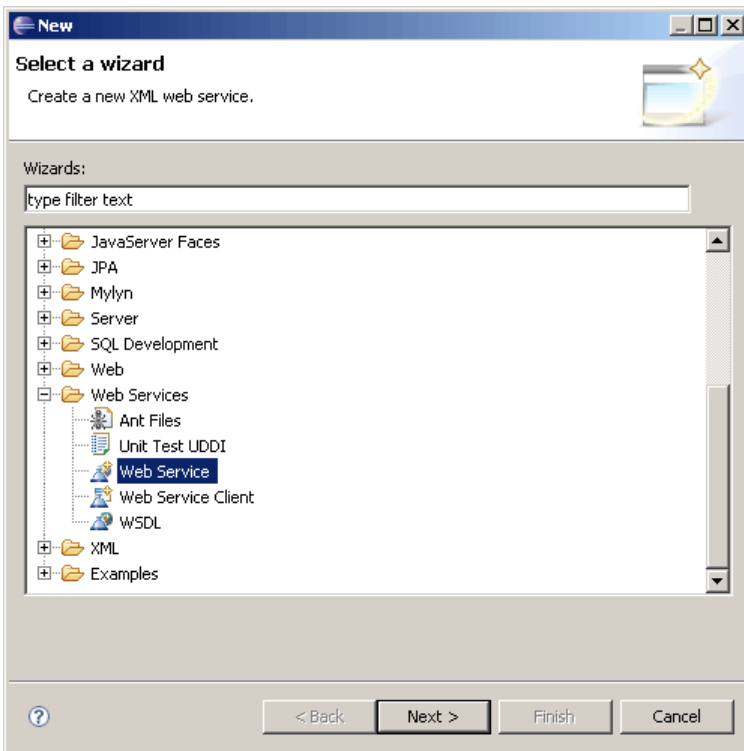


Figure 15 Selecting the Web Service wizard

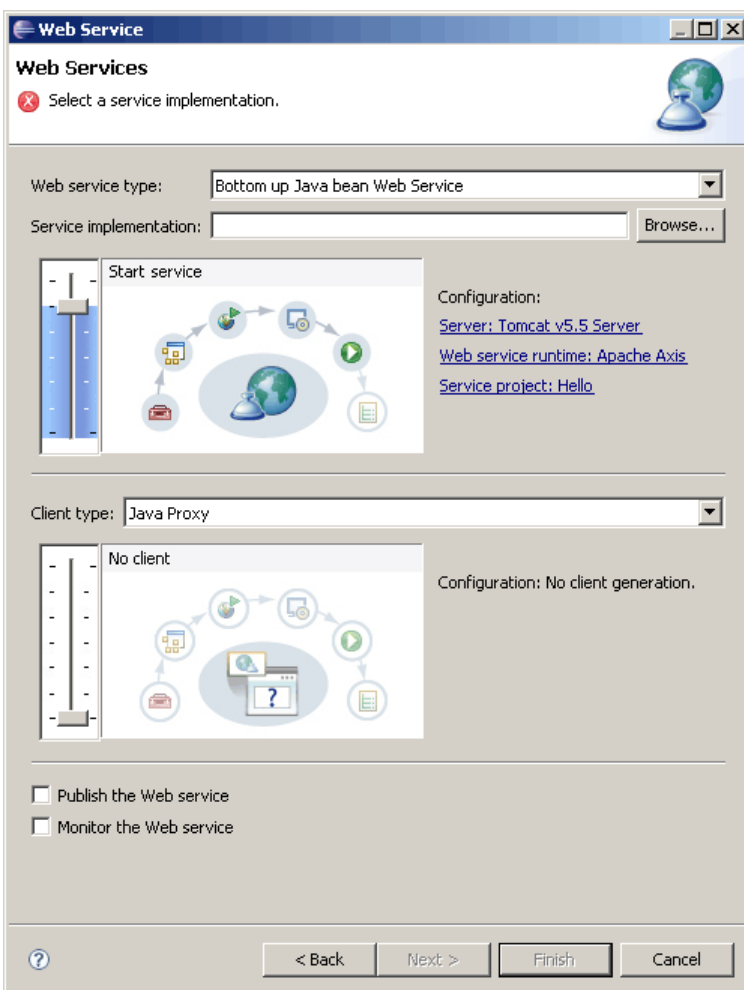


Figure 16 Initial web service configuration

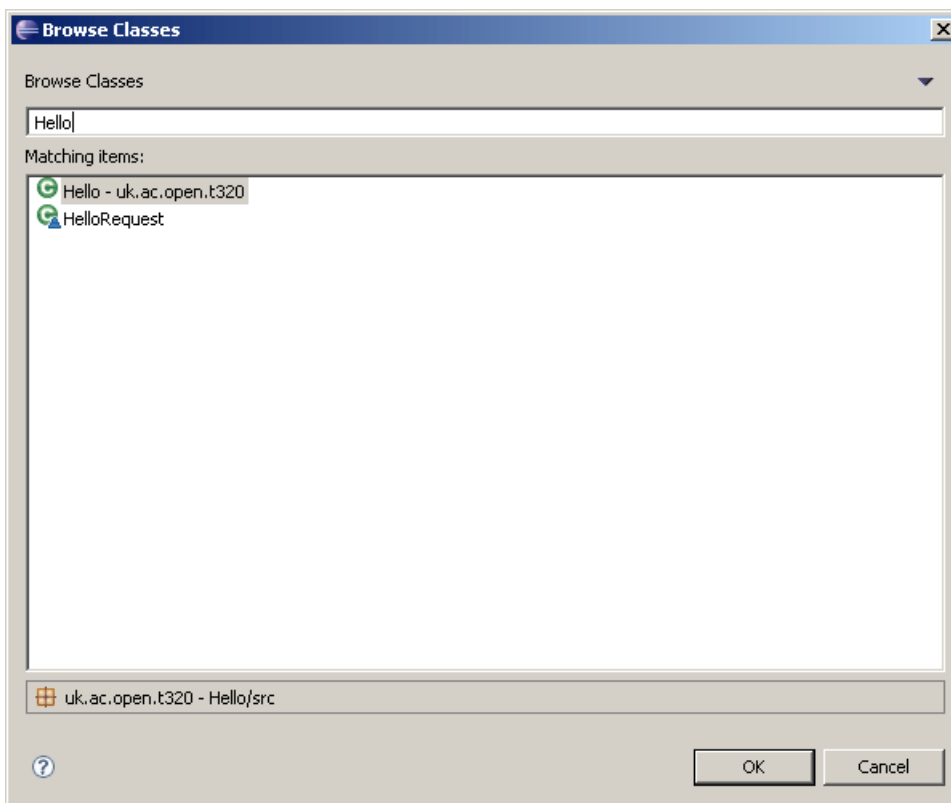
The upper section relates to the web service that you are creating. It names the class you are using and lists the server etc. you are going to use. Notice that the 'Web service type' is given as 'Bottom up Java bean Web Service', which indicates that you are producing a web service from the bottom up based on some Java code (our 'Hello' class). To the left is a slider control, which initially is set near to the maximum level with the text beside it saying 'Start service'. This slider controls what action Eclipse will take in terms of assembling, deploying, starting, etc. the web service.

The middle section allows you to specify that a client should be generated so that you can test the web service. Again this has a slider control, which can be used in the same way as that for the web service. This section is actually the same as you would have encountered had you chosen to create a new 'Web Service Client' in Figure 15.

The lower section contains two check boxes, which allow you to select whether the web service is to be published (to UDDI) or monitored.

As you selected the 'Hello.java' file before starting the web service wizard, it is assumed that this file will be the service implementation. If you didn't select this file then you will see that at the top of the dialogue box there is an error message 'Select a service implementation' (as shown in Figure 16), which will disappear once a service is selected.

If you see this error message then click on the 'Browse' button next to 'Service implementation'. This will bring up the dialogue box in Figure 17. If you start to type the first few characters of the class name 'Hello', you will see that Eclipse lists the classes that match the characters typed.



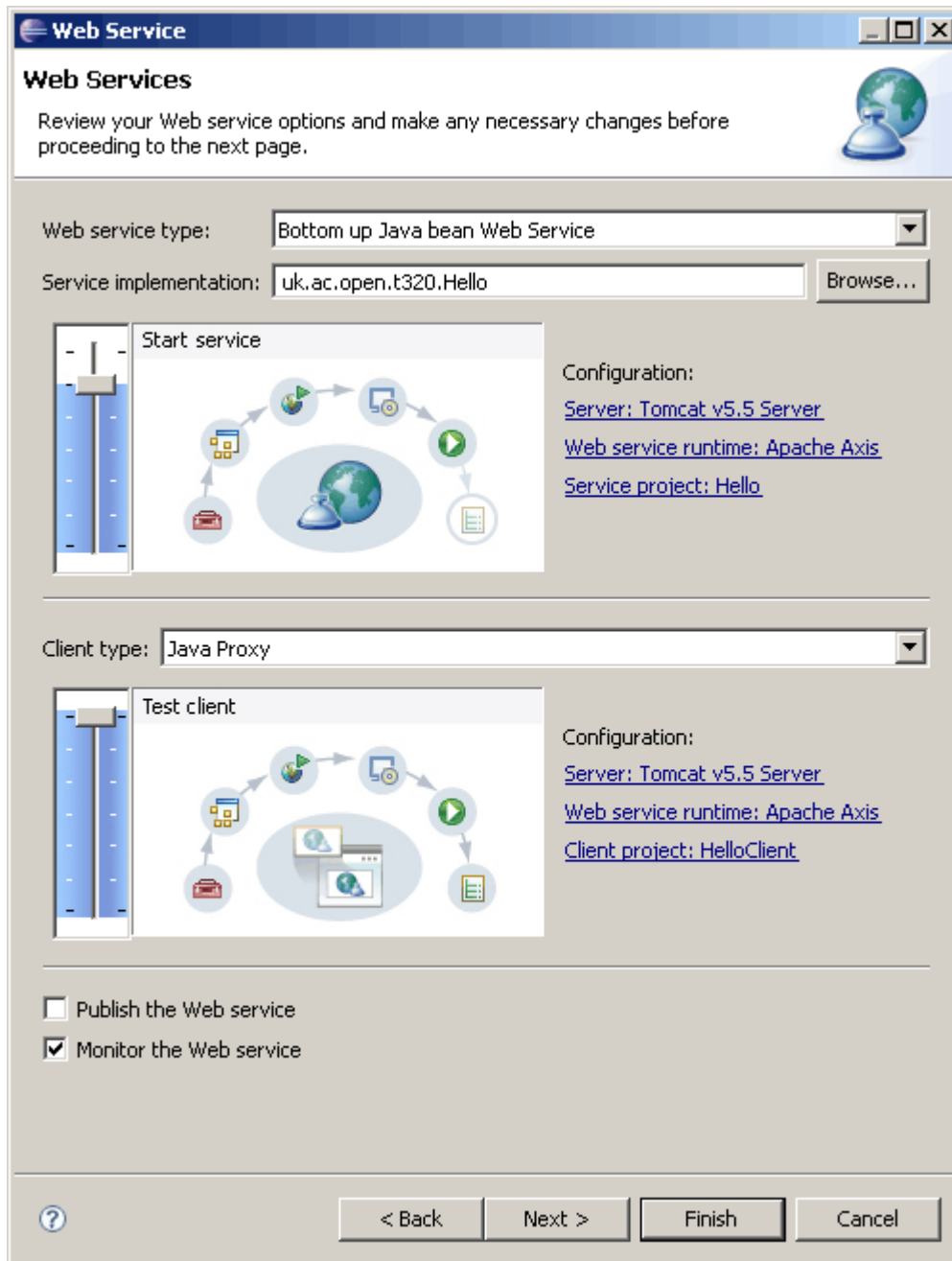
**Figure 17** Selecting the service implementation class

In this case browse to and select the 'Hello' class in the package that you specified earlier. Your package name should include your OUCU as described earlier. Then click 'OK' to return to the Web Service configuration dialogue.

You can now set the slider controls. The slider control for configuring the web service can be left at the 'Start service' setting. Note that the 'Configuration' to the right of the slider specifies the Tomcat server you set up earlier and the 'Hello' project. You will be

introduced to Axis later, but for now you can think of it as a library used by Tomcat to support the running of web services.

The slider control for the client should be set to the 'Test client' level (Figure 18), so that Eclipse generates a client with which you can test the service. Notice that the 'Client type' is listed as 'Java Proxy'. The Java proxy is a Java class that is generated as an application to implement the functionality of the test client. It will be automatically generated and started as a web application running on Tomcat.



**Figure 18** Web service configured

Before clicking 'Finish' tick the 'Monitor the Web service' checkbox. This will instruct Eclipse to start the TCP/IP Monitor, which will display the messages that are sent between Eclipse (via the proxy) and the web service. When you test the service you will then be able to view these XML messages.

When you do click 'Finish', Eclipse will start to create and deploy the web service and make the client for testing the service. These actions may take some time, so you may need some patience, although Eclipse will report the progress made.

If you are running Eclipse for the first time when also starting a Java process then you will usually see the dialogue box in Figure 19 warning you that Eclipse features are being blocked. You should click 'Unblock' and allow Eclipse to run the Java code.



Figure 19 Windows Firewall blocking Eclipse

When Eclipse has finished producing the web service and client, the configuration dialogue box will close and you will see that two new items have been added to the list in the Project Explorer (Figure 20).

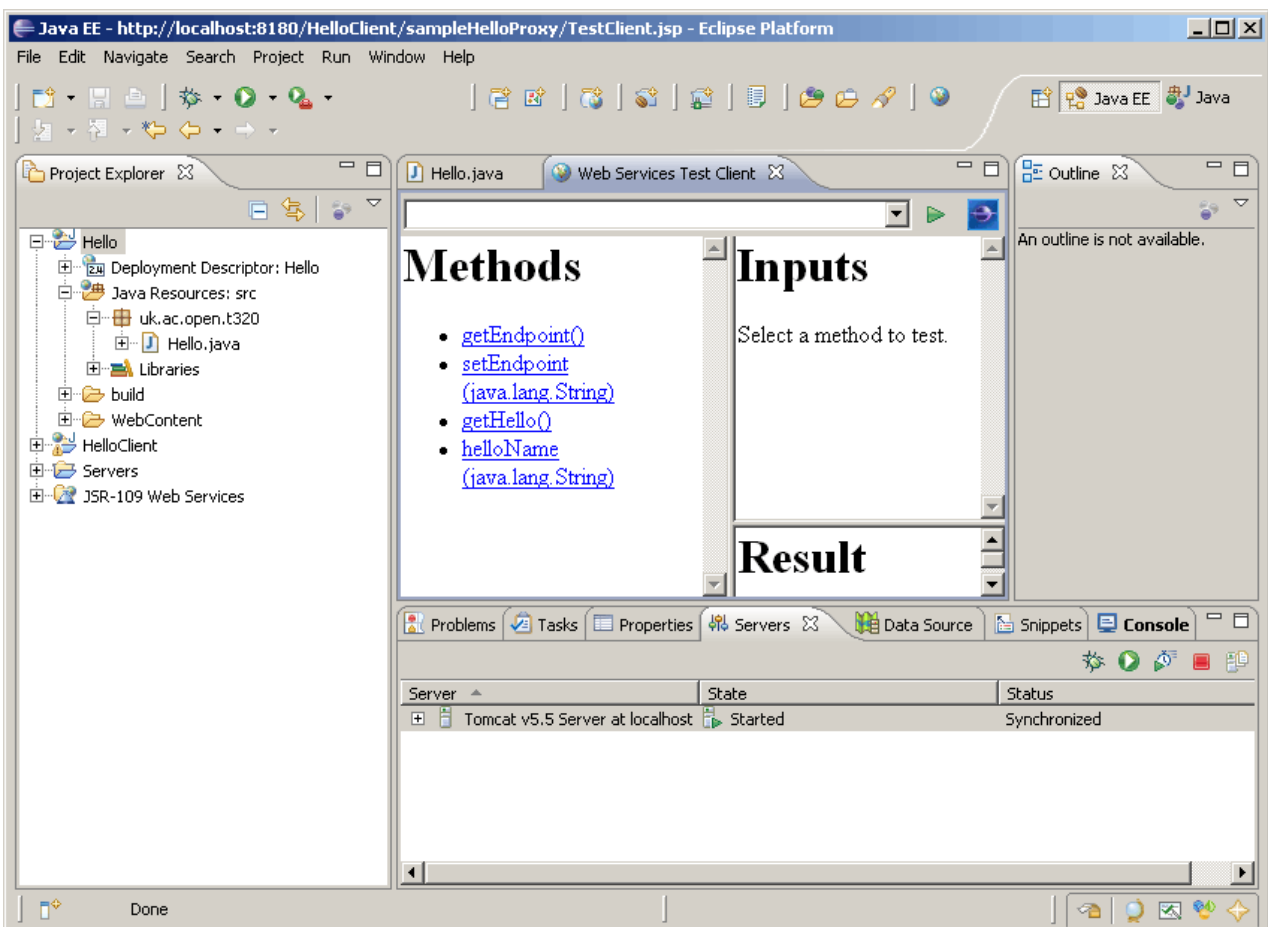


Figure 20 Eclipse with generated client



The first is a Java project called 'HelloClient', which Eclipse created to hold the code for the client that has been generated.

The second is a new folder called 'JSR-109 Web Services'. This folder is intended as the location for any files generated that support the Java Specification Request (JSR) 109 (see <http://jcp.org/en/jsr/detail?id=109> if you want further details). JSR-109 is a Java standard approach for encoding web services within the Java EE environment, but I shall not be concerned with this approach here as it would require an in-depth examination of Java techniques. The folder is actually empty and will not be used in this activity.

Eclipse will also have opened a web browser with a tab named Web Services Test Client. The web browser displays the client web pages as a frameset of three pages with headings 'Methods', 'Inputs' and 'Result'.

At the bottom of the Eclipse workspace you will also see that the Console tab is displaying several messages. These are messages concerning the Tomcat server and its status. Most of these messages are simply for information ('INFO'). It is important that there should be no errors reported; if there are errors reported then there may be a problem with your Tomcat installation. Warnings are not usually a problem, but if you see any errors then you should discuss these with your tutor.

## Testing the web service

### Using the client

You can now test the web service. The Methods pane contains a list of the actions that the web service can be asked to perform. The first three listed, 'getEndpoint()', 'setEndpoint(java.lang.String)' and 'getHello()', are actions that Eclipse generates by default so that certain aspects of the web service can be managed using the test client.

A web service endpoint in this case is the location (a URL) where the web service can be accessed on a network. A similar web services acronym, EPR, refers to the 'endpoint reference' of a service. You may also encounter terminology such as 'SOAP endpoint'. Again this refers to the network location where the web service can be accessed, in this case by sending a SOAP message to the web service.

I shall look at these first three methods later on, but for now you can ignore them.

The last method, 'helloName (java.lang.String)', is the method defined by the Java code of the web service that you created earlier. If you click on this method in the Methods pane then you will see that the Inputs pane changes to display a text box labelled 'name'. All you need do is type a name into this text box, such as 'Nick' (Figure 21), and then click the 'Invoke' button. This will send the text you have entered to the web service via the proxy that Eclipse has generated and started.

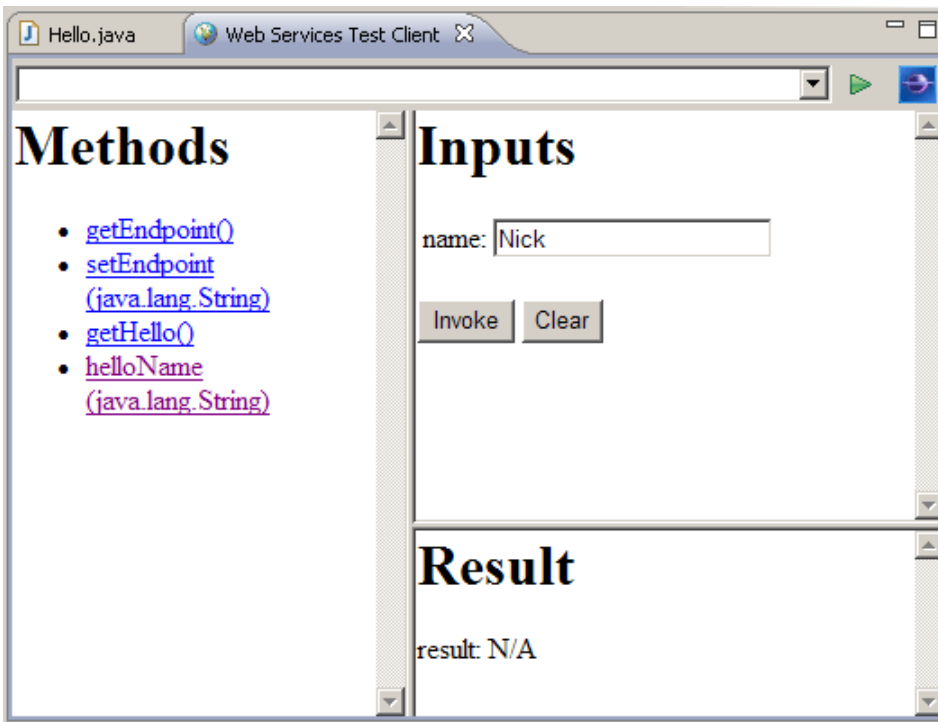


Figure 21 Calling the 'helloName' method

The web service will send a response, which is displayed in the 'Result' pane in Eclipse (Figure 22). At this point the TCP/IP Monitor view should open automatically.

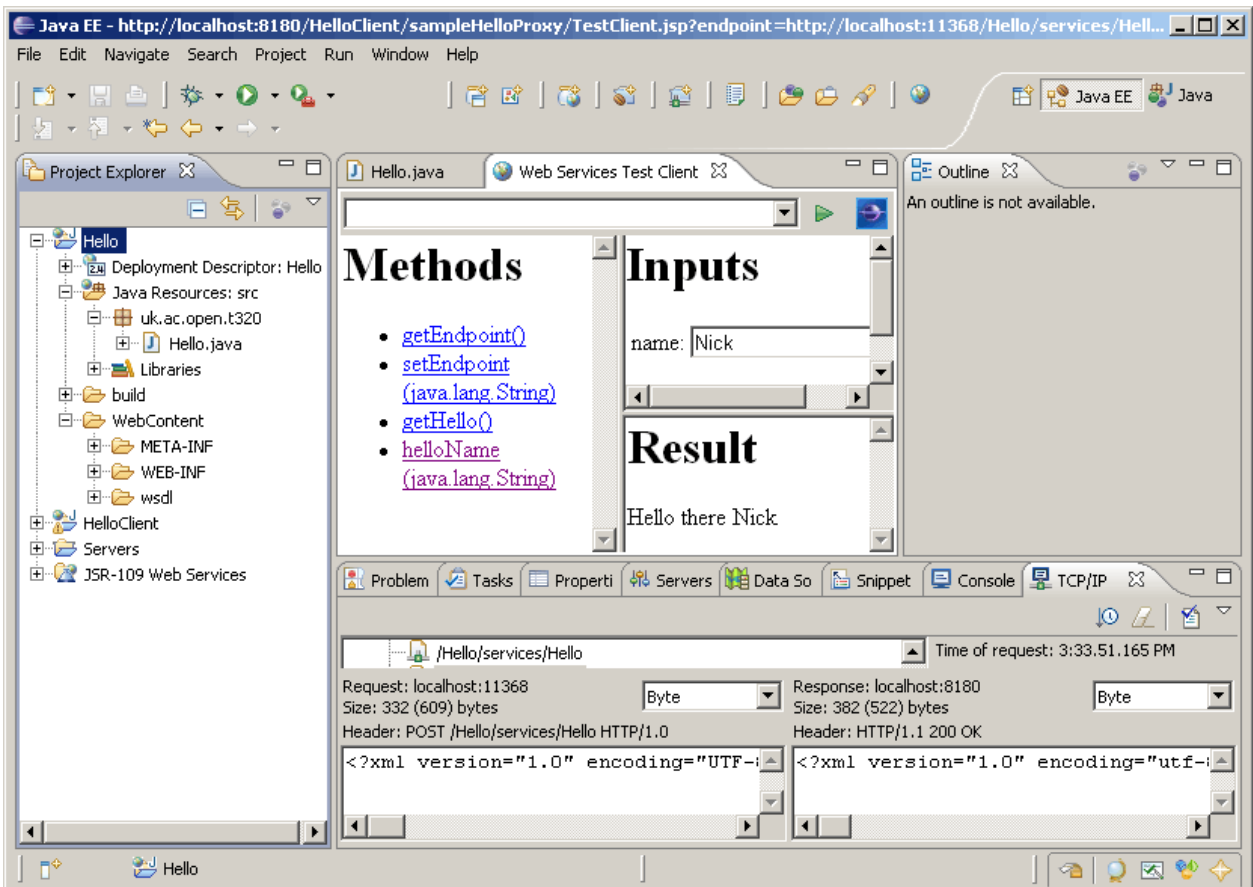


Figure 22 Web service result

## Monitoring web service messages

At the bottom of the Eclipse window, you will see that both the request put to the service and the response from the service are displayed in the TCP/IP Monitor. Eclipse displays these messages all on one line, but here is the request laid out more usefully:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <helloName xmlns="http://t320.open.ac.uk">
      <name>Nick</name>
    </helloName>
  </soapenv:Body>
</soapenv:Envelope>
```

You can see that this is a SOAP envelope with a fairly simple body element. The application's own namespace is constructed using the Java package of the Java class. If a proper unique Java package name was used then this is acceptable as the URI.

The Java method name 'helloName' has been used as a SOAP tag enclosing another tag 'name', which is the name of the variable argument to the 'helloName' method.

These conventions, of course, are restricted to uses of Java but other programming languages can use similar practices.

The SOAP response is very similar, with only a slightly different body element:

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <helloNameResponse xmlns="http://t320.open.ac.uk">
      <helloNameReturn>Hello there Nick</helloNameReturn>
    </helloNameResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

The web service has used a simple convention of appending 'Response' to the request tag 'helloName' and has appended 'Return' to this to label the method's returned result.

## Service descriptions

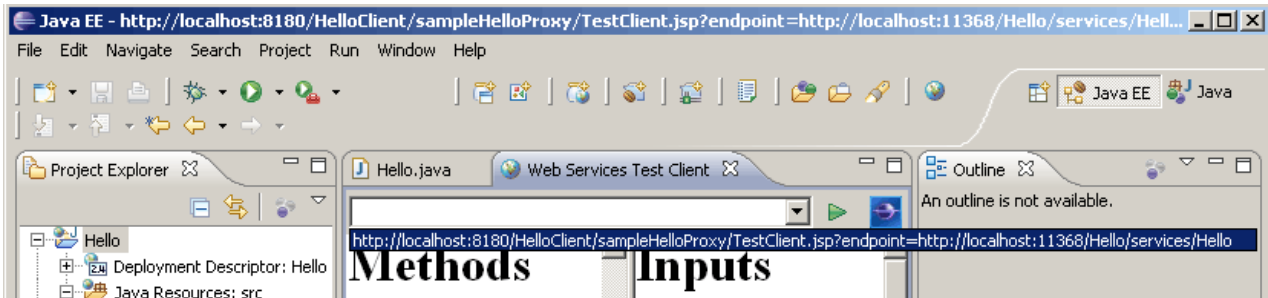
### Web service endpoint

You have now completed the main steps for creating the simple 'Hello' web service. Before you shut down Eclipse, I want to take a quick look at some of the items that have been generated around the 'helloName' method you have invoked.

Earlier I noted that the Methods pane in Eclipse lists three methods other than the 'helloName' method that you created (see Figure 21). The endpoint of the web service is the URL where the service resides and can be accessed. If you click on 'getEndpoint()' and then on the 'Invoke' button, the URL will be displayed in the Result pane as `http://localhost:8180/Hello/services/Hello`.

Note: the port numbers used, such as 8180 in this case, may not be the same on your machine. Thus Eclipse may show different numbers in this and other cases illustrated in this document.

If you typed this URL into a browser you would be disappointed, because although the service resides at this location, the interface you are using is not available at this address. If you look at the value in the URL pull-down at the top of the Web Services Test Client pane (Figure 23) then you will see the actual HTTP URL used to display the test client. This URL is also shown in the top bar of the Eclipse window.



**Figure 23** Display of the test client pages URL

In this case the full URL is:

```
http://localhost:8180/HelloClient/sampleHelloProxy/TestClient.jsp?
endpoint=http://localhost:11368/Hello/services/Hello
```

The URL is that of a Java Server Page (JSP) called 'TestClient.jsp', which includes the frames you are seeing (Methods, Inputs and Result). A JSP web page is simply a web page with embedded Java code to provide additional functionality over HTML. Here, for example, the JSP code is taking the parameter off the end of the URL and using this to access the web service, passing over the SOAP message to the service. Other languages provide similar functionality.

At the end of the URL is a parameter that is passed to this JSP. The parameter is the URL of the endpoint, which is used by the JSP to actually access the web service when making a request. The URL

`http://localhost:11368/Hello/services/Hello` is the URL for the web service via the proxy (the proxy is a Java class itself).

Copy the URL from the Web Services Test Client to your favourite web browser. Make sure that you leave Eclipse running as it was when you called the web service, so that Eclipse is still running the proxy and web service on Tomcat. You should see the same client pages displayed in the browser and be able to access the web service in the same way (Figure 24).

These client pages are very simple, but it should now be obvious that there is little difference between the service you have implemented and tested within Eclipse and a web service that is generally accessible.

We could take the JSP (web pages) generated for the client as a basic template for our own web pages and, although we have to run a proxy application, we could move the code onto a standalone Tomcat and run our 'Hello' application outside Eclipse. However, we will not do that as the proxy approach is a little clumsy for a web application outside Eclipse, and we can quite simply move the application to run on Tomcat without Eclipse support.

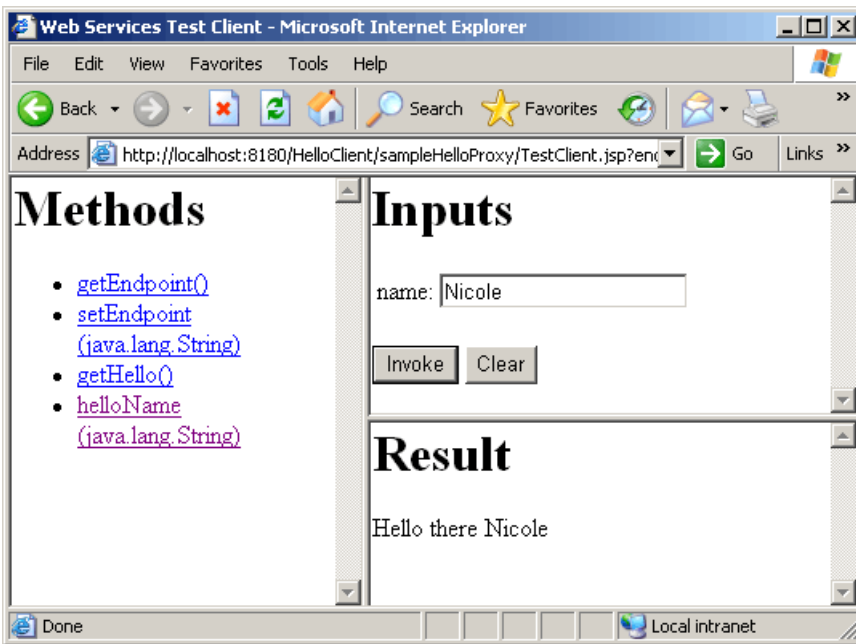


Figure 24 Eclipse client use in browser

## Axis and WSDL

The Eclipse generation of a web service uses a commonly available library of code called Axis. You will see more of Axis shortly, but for now it's useful just to start to get a feel for what it is providing.

If you use the actual web service URL (<http://localhost:8180/Hello/services/Hello>) in a web browser, or go via the proxy (<http://localhost:11368/Hello/services/Hello>), you will see that a web page is generated that says there is an Axis service present and that a form to access the service might be put there (by you) later (Figure 25).

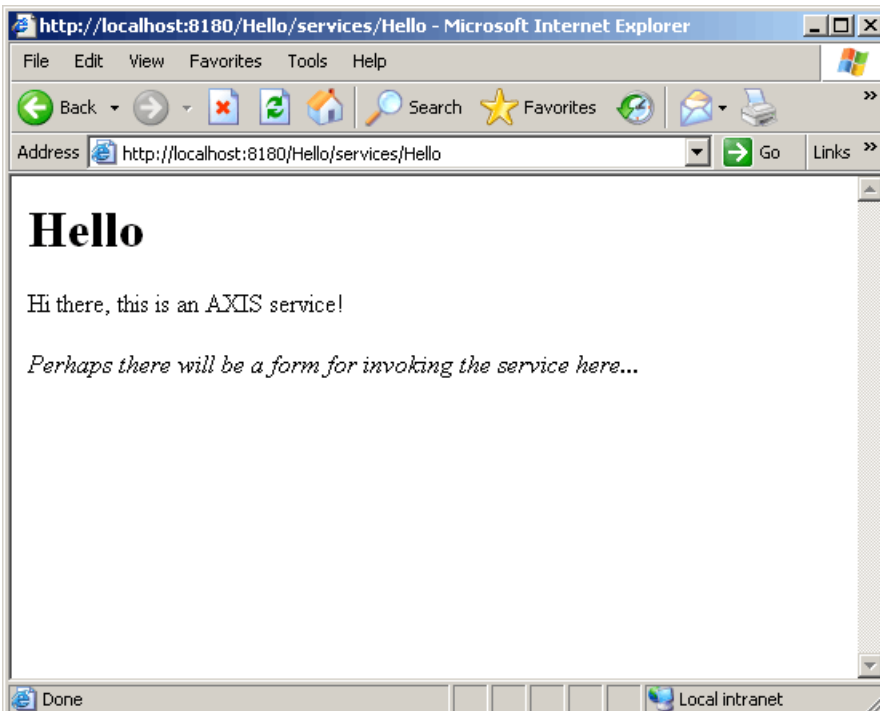
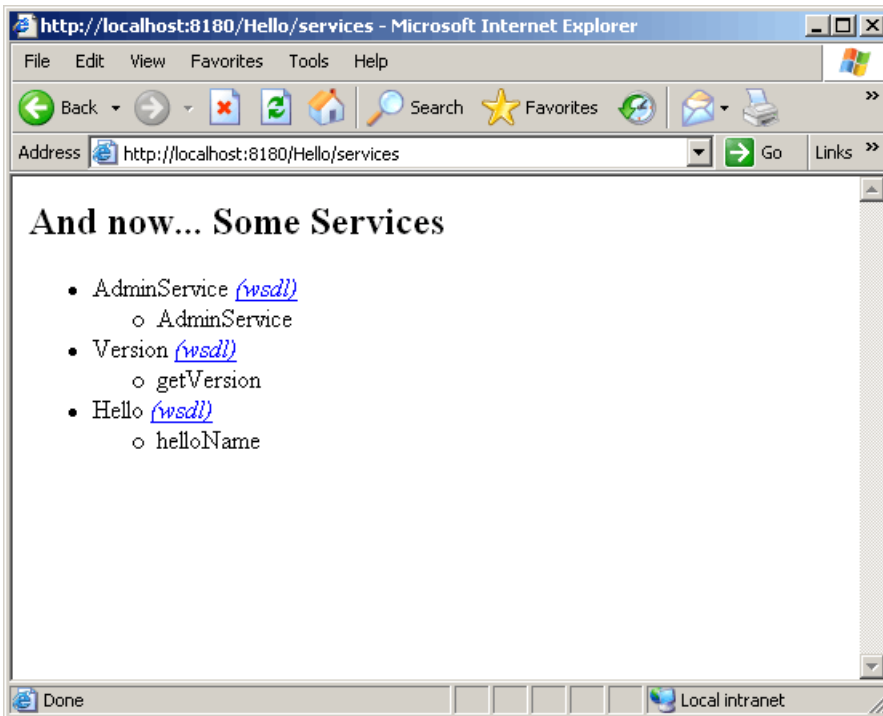


Figure 25 Endpoint for the 'Hello' web service

Whilst Axis generates the page that says that a service is present, it doesn't actually say anything about the service. However, Axis also generates a page that lists all the web services it has available and the WSDL document for each web service. These are available from URLs related to the web service you have deployed. For instance, the list of services is given at <http://localhost:8180/Hello/services> (Figure 26), which is simply the URL of our web service truncated to the 'services' level. You should see a page listing two default Axis services, the 'AdminService' and 'Version' service, as well as the 'Hello' service you have just implemented. These two default services are distributed 'pre-built' with Axis inside Tomcat.



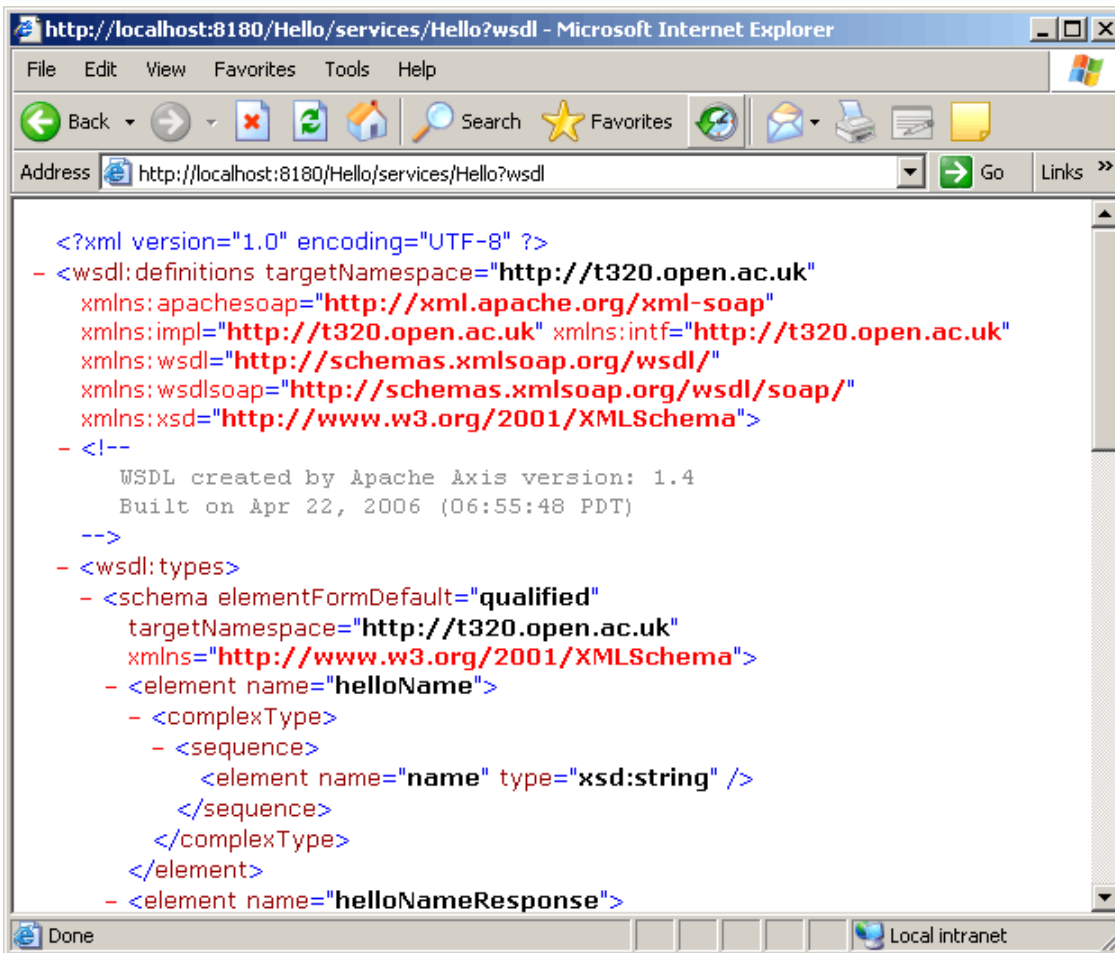
**Figure 26** Axis-generated list of web services

It may have caught your attention that whilst Axis doesn't provide web pages to actually access web services, it does provide access to the WSDL document that it has generated for each service. Click on the '(wsdl)' link next to the 'Hello' service. You should see the complete WSDL document for this service, which Axis has generated (Figure 27).

The complete listing of this WSDL document is given in Appendix 1. Don't worry about this for now, but you will need to use it later.

What's more important than the precise details, as you have seen, is that tools like Eclipse and Axis provide a very good level of support for the development and description of web services. Take the case of the simple 'Hello' web service described here. With the help of Eclipse and Axis, you have:

- created the logic of the service and deployed this to the application server
- created a simple client and used this to test the service
- examined the SOAP message exchange
- generated the WSDL that could be used by other clients to use the service.



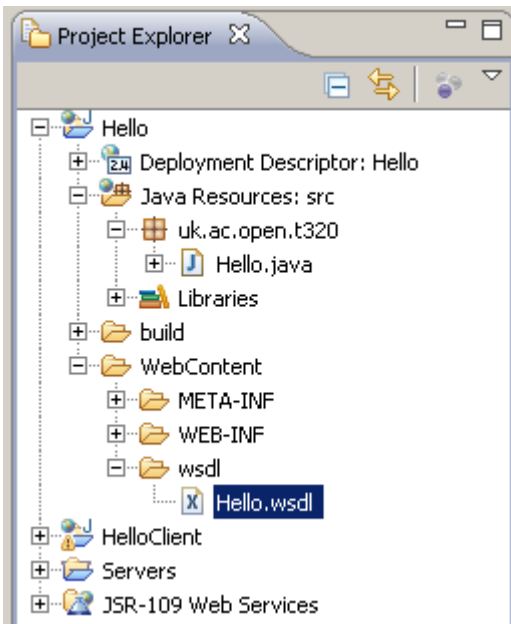
**Figure 27** Axis-generated WSDL document for the 'Hello' web service

If you return to the description of web service architecture in Part 1, and look again at Figure 5, you will realise that only two major challenges remain for you to have created a fully fledged web service. These two steps are as follows:

- 1 Deploy and run the web service directly on an application server (such as Tomcat) rather than deploying and running the service with the aid of Eclipse and a proxy application.
- 2 Publish a description of the web service in a directory service (UDDI) so that it can be located and accessed in a proper fashion.

You will shortly be undertaking both of these tasks.

Make sure that you keep your web service project in Eclipse and note where the WSDL file resides, as you will be using this later. It can be found in the 'wsdl' directory of the project's 'WebContent' directory (Figure 28).



**Figure 28** Location of 'Hello.wsdl' file in the 'Hello' project

## Summary

In this activity you have been taken through the steps required to create, deploy (to an application server on your local machine) and test a simple web service. This is a significant step in understanding at first hand how SOA actually works. In subsequent practical activities you will deploy your web service to a server on the Internet, publish its description to a UDDI registry, test its level of interoperability and be able to access and use it on the Internet.

## Appendix 1: WSDL generated for 'Hello' web service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://t320.open.ac.uk"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://t320.open.ac.uk"
  xmlns:intf="http://t320.open.ac.uk"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--WSDL created by Apache Axis version: 1.4
  Built on Apr 22, 2006 (06:55:48 PDT)-->
  <wsdl:types>
    <schema
      elementFormDefault="qualified"
      targetNamespace="http://t320.open.ac.uk"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="helloName">
        <complexType>
          <sequence>
            <element name="name" type="xsd:string" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
</wsdl:definitions>
```



```
</element>
<element name="helloNameResponse">
  <complexType>
    <sequence>
      <element name="helloNameReturn" type="xsd:string" />
    </sequence>
  </complexType>
</element>
</schema>
</wsdl:types>
<wsdl:message name="helloNameResponse">
  <wsdl:part element="impl:helloNameResponse" name="parameters" />
</wsdl:message>
<wsdl:message name="helloNameRequest">
  <wsdl:part element="impl:helloName" name="parameters" />
</wsdl:message>
<wsdl:portType name="Hello">
  <wsdl:operation name="helloName">
    <wsdl:input message="impl:helloNameRequest" name="helloNameRequest" />
    <wsdl:output message="impl:helloNameResponse" name="helloNameResponse" />
  </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="HelloSoapBinding" type="impl:Hello">
  <wsdlsoap:binding
    style="document" transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="helloName">
    <wsdlsoap:operation soapAction="" />
    <wsdl:input name="helloNameRequest">
      <wsdlsoap:body use="literal" />
    </wsdl:input>
    <wsdl:output name="helloNameResponse">
      <wsdlsoap:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="HelloService">
  <wsdl:port binding="impl:HelloSoapBinding" name="Hello">
    <wsdlsoap:address
      location="http://localhost:8180/Hello/services/Hello" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```