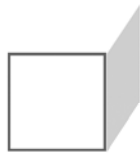
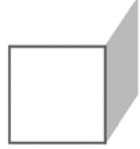
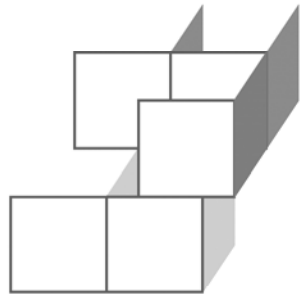




CHAPTER 7
**WEB SERVICES DESCRIPTION
LANGUAGE**



OBJECTIVES

After completing “Web Services Description Language,” you will be able to:

- Describe the role of WSDL in providing type information for Web services.
- Describe the benefits of providing full type information for a Web service.
- Describe the WSDL service-description model.
- Write WSDL documents to describe message types and service types.
- Create bindings for a service description to a specific protocol, such as SOAP.
- Generate client code for a WSDL-described service.

Component-Based Software

- A clear pattern has evolved for infrastructure that supports distributed computing via **components**.
 - Such architectures support what is known as **distributed object computing**, or **DOC**.
- There are several established expressions of this pattern: CORBA, DCOM, and EJB are the most well-known.
- All of these architectures offer the following features:
 - Separation of **interface** and **implementation**
 - Objects offer one or more **interfaces**
 - Interfaces are composed of **methods**
 - Methods have **signatures** of **parameters** and **returns**
 - **Strongly typed** at a fine granularity, down to parameter and return types
 - Support for **exceptions** and distributed exception handling
 - Support for **remote invocation** of these distributed objects, via the Proxy pattern – this is possible because of the interface/implementation split

Component Models

- The target domains of popular DOC models vary:
 - EJB is language-dependent
 - DCOM is platform-dependent
 - CORBA is fully portable
- Each of these models defines the following as a means of supporting the aforementioned feature set:
 - A **type model** for RPC definitions, including low-level definitions such as how many bytes are used to express an integer, and in what order they are transmitted
 - A **wire format** and **transport** for RPCs (DCE, IIOP, the Java RMI wire protocol and transport for EJB)
 - A **runtime** to support components (the COM libraries and service managers, the ORB, the EJB server/container)
 - An **interface definition language** (MIDL, CORBA IDL, Java itself)

Web Services as Components

- SOAP fosters the development of **component-based** Web services, extending the DOC model.
 - A SOAP-based service can be thought of as an API exposed over the Web and addressable at a single URI, which becomes a sort of portal for RPC interactions.
 - Transfer of documents is also supported, but for the moment we will focus on RPC-style interactions.
- The Web services architecture can be seen to fit the pre-existing pattern for DOC architectures:
 - Its target domain is the **Web**, which is broad indeed: no specific platform, no specific implementation language
 - **XML** is the basis of the type model; SOAP refines this and builds on it, and XML Schema is the most likely ultimate solution
 - The wire format is **SOAP**
 - **HTTP(S)** is the transport (other options include SMTP/POP/IMAP and FTP)
 - A SOAP-aware **Web server** acts as the runtime ...
- What, or where, is the IDL for Web services?

A World Without Type Information

- **In a hypothetical world blessed only with HTTP, XML, and SOAP, there is a clear need for a Web-services IDL.**
 - We've exposed this as a critical piece of the Web services architecture based solely on comparisons to DOC models.
 - We will now look at some of the concrete advantages of rich type information, and of complete descriptions of Web services.
- **Consider a Web service built using SOAP 1.1 over HTTP.**
- **Any application will do for an example; let's hypothesize a service that offers to shop around for home-equity loans on behalf of the client.**
 - The service supports one message type, by which the client provides the address and assessed value of the property, amount of debt related to any liens on the property, and requests a loan or credit line of a certain size.
 - The response provides up to three qualifying creditors, each listed by name, interest rate and other terms they're offering.
- **Let's assume that the service is up and running, implements SOAP 1.1 perfectly, has no availability problems, and is at a known, advertised URL on the Web.**

The Need for Service Description

- Consider some of the problems, or challenges perhaps, involved in dealing with this service.
 - For each, we’ll talk about a possible solution and the level of descriptive support that would be needed.

Problem or Challenge	Solution
<p>A developer who wants to write a client of the service doesn’t know how to invoke it!</p> <p>What are the XML element names that should be used for the various arguments?</p>	<p>This can be learned by example – a look at a sample request and response. i.e. inductive learning.</p> <p>A schema for each would be a more reliable solution, though: allows deductive learning.</p>
<p>When bad messages come in, how can the service reject them efficiently and clearly?</p>	<p>A schema that precisely defines the content model for each message could be used to validate incoming messages.</p>
<p>Client development is laborious, as someone has to write application code to build the SOAP request and parse the SOAP response.</p>	<p>Now a schema is not enough.</p> <p>A descriptor identifying certain parts of the message as method names and arguments could be used to generate a client-side stub that would handle the details of SOAP messaging.</p>
<p>Adding intermediaries such as security and resource managers to the system is difficult because they don’t understand this particular message type.</p>	<p>A descriptor would allow such intermediaries to dynamically interpret the SOAP messages and then perform their tasks.</p>

An IDL for Web Services

- The previous example highlights some of the facilities that can make Web-service and client development easier and can make running services more robust:
 - **Validation** of message content at runtime – this can save both service and client a lot of time and trouble, by pushing errors to the front of the process and by producing clear error messages about invalid content.
 - **Dynamic** service and **invocation** at runtime – SOAP intermediaries such as firewalls, resource managers, filters, and routers can act intelligently on messages if they are given clear metadata.
 - **Code generation**, prior to runtime, to support static stubs and skeletons, and other dedicated components that support, use, or extend a Web service – more generated code means less maintenance, too, even across the service-client boundary
- If only we had a “Web-services IDL” at our disposal! This utopian language could solve all these problems.
 - It would have to go beyond a schema language, though: it would have to recognize the patterns prevalent in SOAP messaging: common structures and behaviors.
 - For instance, this language would have to identify which XML elements were method names, and which were arguments to methods, or return values.
 - It would have to organize request and response messages into something akin to a method signature on a Java class.

Web Services Description Language

- We can drop the hypothetical “WS-IDL” at this point, because there is a real language in current use that solves the same problems.
- This is the **Web Services Description Language**, or **WSDL**.
 - WSDL 1.1 is a Note to the W3C, and future versions will be slated as W3C recommendations.
- WSDL is an XML vocabulary for describing Web services. A WSDL descriptor defines:
 - SOAP message types and their valid content, including supporting data types
 - **Operations**, which can include one or more possible message types in a **scenario** such as one message type for a request and one for a response
 - Services themselves, as components that support one or more operations at defined endpoints (URIs)
- Operations are further grouped into **port types**, which are roughly analogous to interfaces in other DOC models and some programming languages.

WSDL Namespaces

- The WSDL specification defines several namespaces for use in descriptors.
- The primary WSDL namespace URI is shown below; the common prefix is `wsdl:`, although descriptor documents often set this namespace as the default.

`http://schemas.xmlsoap.org/wsdl/`

- Each **binding** of WSDL to a possible messaging protocol is given its own namespace.
 - We'll look at the concept of bindings a bit later.
 - The SOAP binding namespace, commonly prefixed `soap:`

`http://schemas.xmlsoap.org/wsdl/soap/`

- The HTTP binding namespace, commonly prefixed `http:`

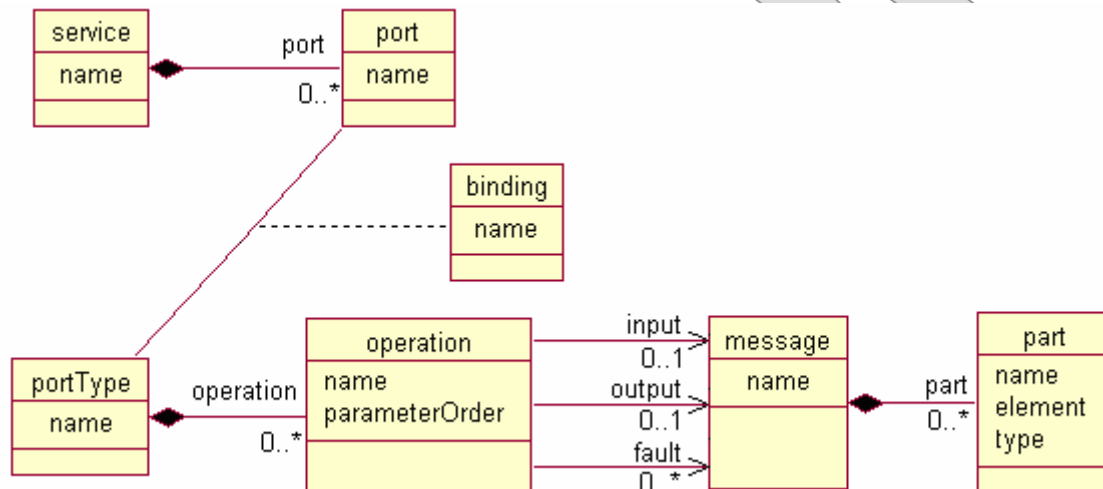
`http://schemas.xmlsoap.org/wsdl/http/`

- The MIME binding namespace, commonly prefixed `mime:`

`http://schemas.xmlsoap.org/wsdl/mime/`

The WSDL Description Model

- WSDL is an XML vocabulary that uses seven main element types to describe a Web service:



- All UML attributes map to XML attribute names.
- **Services** are modeled as collections of **ports**.
- Each **port** has a **portType**.
 - The port type is much like an interface declaration in COM, CORBA, EJB or an OO programming language: it consists of one or more method signatures.
 - These are decomposed into **operations**, **messages**, and **parts**.
- The **port** is **bound** to its port type for purposes of messaging over a specific protocol.
 - Thus the port type is protocol-independent.
 - The **binding** element provides details of how messages can pass over a given protocol.

A WSDL Document

EXAMPLE

- Here is a fairly simple WSDL document – find this in **Examples\Math\JAX-RPC\WSDL\Math.wsdl**.

```
<?xml version='1.0'?>

<definitions name='Math'><!-- Namespaces elided.-->
  <types/>
  <message name='Math.Request'>
    <part name='x' type='xsd:double' />
    <part name='y' type='xsd:double' />
  </message>
  <message name='Math.Response'>
    <part name='return' type='xsd:double' />
  </message>

  <portType name='Math'>
    <operation name='Add' parameterOrder='x y'>
      <input message='tns:Math.Request' />
      <output message='tns:Math.Response' />
    </operation>
    ...
  </portType>

  <!-- We'll look at the binding later. -->

  <service name='MathService'>
    <port name='Math' binding='tns:MathBinding'>
      <soap:address location=
        'http://LOCALHOST:8080/Math/JAX-RPC/Calculate' />
    </port>
  </service>

</definitions>
```

WSDL Descriptors as Schema

- A WSDL descriptor is not a schema by the definitions of the W3C's XML Schema recommendation.
 - It can reference XML Schema documents to define XML types for parts of message content.
 - It can embed an `xs:schema` element to define its own types.
 - It does not itself define types that could be used directly by an XML-Schema-aware validating parser.
- A WSDL descriptor is a sort of schema, however.
- It defines two models that can be used to generate or to validate messages:
 - An **abstract model** of services, port types, operations, etc.
 - A **concrete model** which binds the abstract model to a specific messaging protocol such as SOAP.
- A WSDL descriptor defines a **target namespace**, just as an XML Schema does.
 - Definitions in the rest of the descriptor are said to **populate** that namespace – much as XML Schema components do.

The Schema for WSDL Descriptors

- Since WSDL is a well-defined XML vocabulary, there is an XML Schema that sets all the rules of this vocabulary, and can be used to validate WSDL documents.
 - The schema can be found in the WSDL 1.1 specification.
- To validate a WSDL document is a straightforward exercise, given a validating parser and the schema for WSDL and its several related namespaces:
 - The relevant binding, for instance the SOAP binding.
 - The SOAP and SOAP encoding schema.
- Tools that use WSDL documents to generate application code routinely begin their work by validating the WSDL input.

Associations Between Components

- The earlier UML diagram shows that the WSDL model uses both **composition** and **association**.
 - A **portType** is **composed** of **operations**. This is implemented by normal XML element composition.
 - An **operation** is **associated** with **messages**. This is implemented by use of an attribute identifying the associated element by its **name** attribute.

```
<operation name='Add' parameterOrder='x y'>  
  <input message='tns:Math.Request' />  
  <output message='tns:Math.Response' />  
</operation>
```

- The type of the referenced element is implicit based on the referencing element type and the attribute name.
- WSDL components define types, and so they are referenced by **qualified names**.
 - The **name** of a component is a local name; it is implicitly qualified by the WSDL descriptor's **targetNamespace**.
 - The referencing attribute, such as **message** in the snippet above, is an explicitly qualified name – note the **tns:** prefix.
 - We cannot expect to reference types in our own namespace implicitly, even though this would seem natural.
 - It must be possible to combine WSDL descriptors, and so **message** above might have referred to something defined in another descriptor, for instance “other:yourMessageType”.

Interface Description

- Let's look at what in DOC-speak we would call **methods** on a Web-service **interface**.
- In WSDL we translate this to **operations** on a **portType**.
 - In other words, a WSDL operation defines a possible interaction between client and service: typically a request and response message.
 - Ideally, the client should see sending the message and receiving the response as simple, local operations.
 - Generated code can handle the actual messaging.
- A key concept of DOC systems is **location transparency**, and what we're discussing here is the extension of that concept to Web services.
 - Messaging may be over a remote connection, but it may just as well be in-process. Generated code can hide these details from client and service code.
 - Just as any IDL will have **mappings** to programming languages, so there will be mappings from WSDL to Java, C#, VB.NET, etc.
- A WSDL operation captures the expected semantics of a SOAP message:
 - The target of the invocation – a specific method name
 - The parameters for that method, by name, type and order

Messaging Scenarios

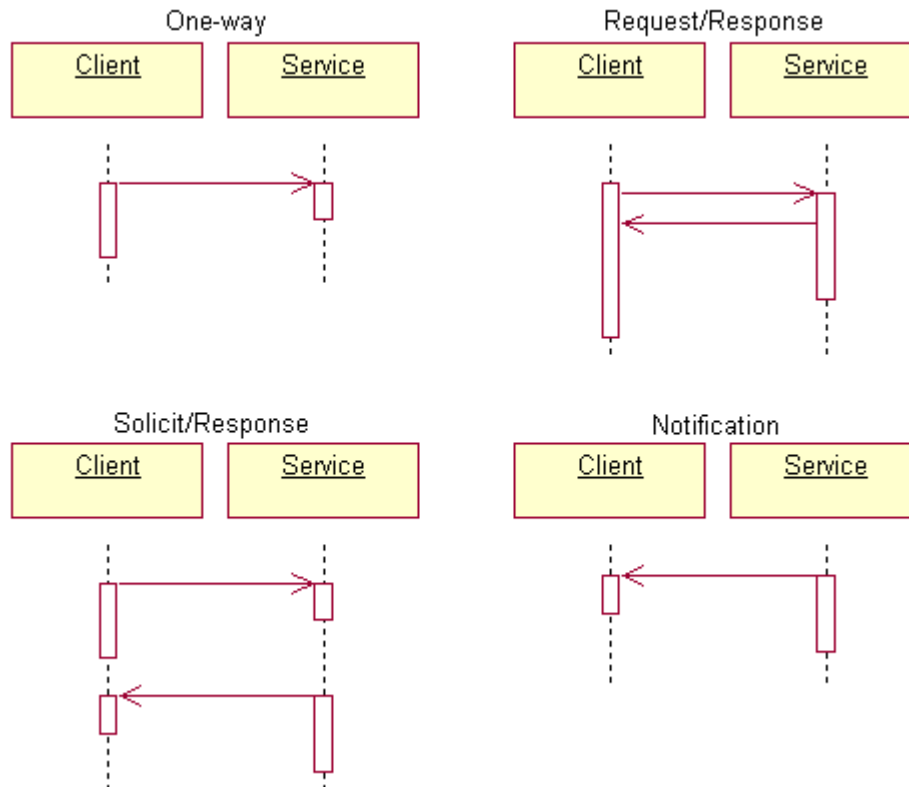
- In most programming languages, a method has zero to many parameters, each of a certain type, and zero-to-one return types.

```
double Add (double x, double y);
```

- Using SOAP, there is a message passed in each direction, capable of carrying an arbitrarily-complex hierarchy of arguments.
- The WSDL specification recognizes four **transmission primitives**, each suited to a different relationship between client and service:
 - One-way
 - Request/Response
 - Solicit/Response
 - Notification
- Each calls for a different combination of messages.
 - For instance the one-way scenario has no use for a response message.

Messaging Scenarios

- The following diagrams illustrate the exact timing of each transmission primitive as defined by WSDL:



- The latter two are not intuitive, and they imply that the “client” can receive requests and is somehow registered with or known to the service.
- These two are really second-class citizens in the WSDL specification.
 - Bindings are only defined for request/response and one-way.
 - The WSDL specification expects that extension bindings would be defined by any specification that calls for use of solicit/response or notification primitives.

Operations: Input, Output, and Fault

- WSDL identifies a job that a service might do for a client using an `operation` element.
 - The WSDL **operation** is analogous to the **method** or **member function** in OO programming languages.
- The WSDL `operation` identifies a different `message` element for each possible message in the invocation.
- Each component message is given a designation as:
 - **input**
 - **output**
 - **fault**
- For example, the request/response primitive calls for two to three of these (and possibly multiple fault types); the one-way primitive requires only an input message.

Messages

- The `message` element then defines the exact information that's expected in a single message, using `part` elements and possibly referencing separate schema that define necessary XML types.

- A `message` may have several `parts`, each of a simple type, as in the example a few pages back.

```
<message name='Math.Request'>  
  <part name='x' type='xsd:double' />  
  <part name='y' type='xsd:double' />  
</message>
```

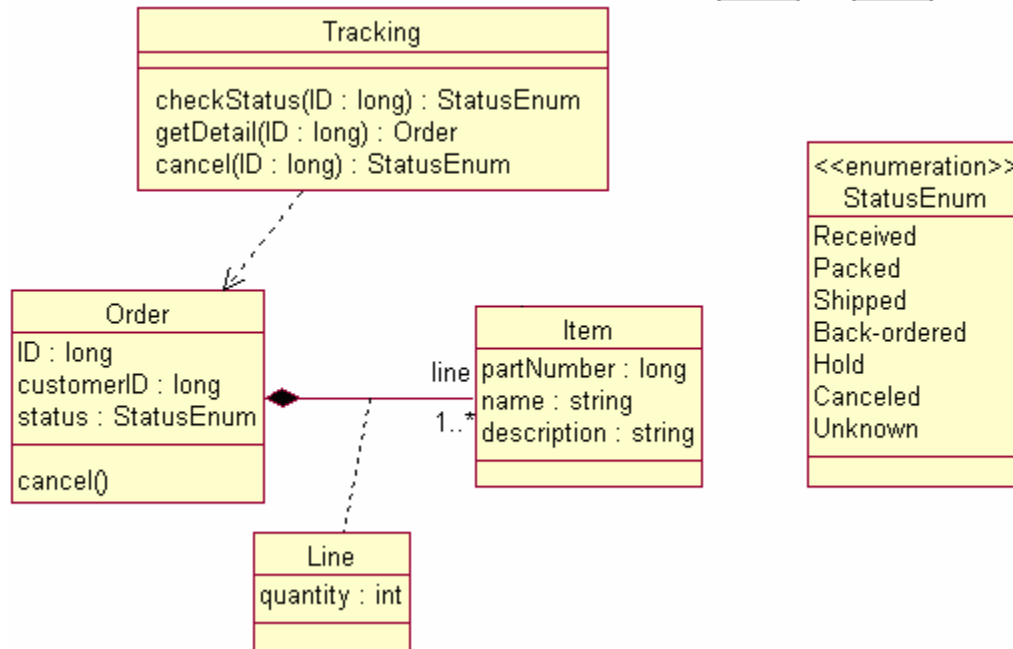
- A `message` may also have a single `part` of complex type. The type must be defined in a referenced schema or in the schema that can be embedded into the WSDL descriptor as a child of the `types` element.

- Message types can be re-used in multiple operations.

- In the earlier example, all five operations have the same semantics: a `Math.Request`-type request and a `Math.Response`-type response.

The Order-Tracking Service

- For some of our lab work in this module we'll build and use various components of an order-tracking Web service:



- The **Tracking** service interface wraps some basic domain logic.
 - **Orders** are composed of one or more **Lines**, each of which in turn defines some quantity of an **Item**.
 - An **Order** also has **status**, which is of an enumerated type.
 - The service is mostly concerned with checking order status and information, but has one mutator method that allows a customer to cancel an order, which of course changes the status field.

Code Generation

- For our labs in this chapter we will need a WSDL-to-Java code generator.
 - The lab starter code includes completed client-side application code which relies on stubs and serializers that have yet to be created.
 - You will write WSDL to describe the target service, and you will know that your descriptor is substantially correct when you can generate code from it that makes the client application work correctly.
- The J2EE RI includes the **wscouple** tool.
 - We will look at this tool in more detail in later chapters.
 - For now, we will rely on one simple usage, which asks the compiler to create client-side source and class files as directed by an XML configuration file (which will be provided for the upcoming lab):

```
wscouple -gen:client -keep -d Build config.xml
```
 - The **config.xml** file identifies the WSDL file and attaches a number of attributes to the process of generating Java code from it.
 - Source and class files generated and compiled from the WSDL will appear under the **Build** directory, as directed by the **-d** option, and further sifted into subdirectories based on their Java packages.

Abstract WSDL for Order Tracking

LAB 7A

In this lab you will build the abstract WSDL model of messages supported by the Order Tracking service. For now, you will validate your descriptor by passing it through the **wscmpile** tool; in a later lab you will use the complete WSDL descriptor to generate service stub classes, which can be used by a prepared client application.

Detailed instructions are contained in the Lab 7A write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

Service Description

- So, if an operation describes how to invoke a method on a component, how is one expected to find the component itself?
- A WSDL `service` element describes a service as a collection of one or more components.
- A `service` includes `port` elements, each of which describes a component class.

```
<service name='MathService'>  
  <port name='Math' binding='tns:MathBinding'>  
    ...  
  </port>  
</service>
```

- At runtime, an instance of this component type is promised to be available at the given port.
- Each `port` has a port type, but this type is identified by reference to a `binding` – we'll look at bindings in a moment.

Extending WSDL

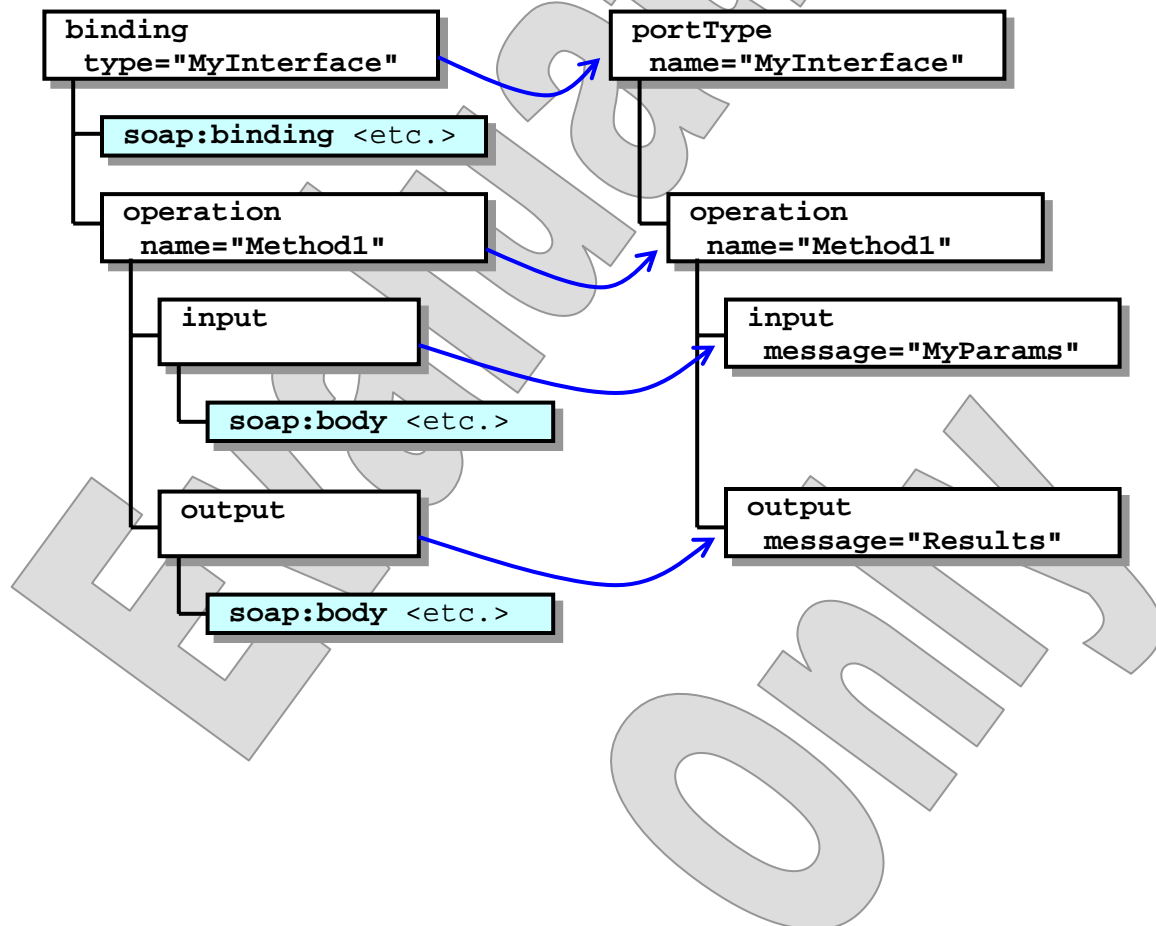
- The components from the WSDL namespace combine to express an **abstract model** of the Web service's semantics.
- The WSDL vocabulary also allows the inclusion of elements from non-WSDL namespaces at certain points in a descriptor.
- These are called extensibility elements, and they extend the abstract model to define a **concrete model** for a particular messaging protocol.
- The WSDL 1.1 specification defines **bindings** for a few common protocols – SOAP, HTTP, and MIME attachments – and others are possible.
 - Each defined binding will occupy its own namespace.
 - Common usage is to set the WSDL namespace as the default and then use prefixes for the namespaces of various binding elements.
- Extensibility elements are allowed to appear by way of an XML Schema base type for WSDL components that includes **wildcards** to permit any content from non-WSDL namespaces.
 - The rules of XML Schema dictate that this content – the extensibility elements – must appear before WSDL components in the child list, not after them.

The Binding Component

- One WSDL component, the **binding**, has the specific purpose of extending the abstract service model.
 - The **binding** element can be referenced by a **port**.
 - In turn, it references a specific **portType**.
 - In this way it connects the two, and in the process it adds protocol-specific information
- The **binding** is the primary means of extension, but not the only one.
 - Extensibility elements can appear elsewhere, as we'll see.

A Structural Pattern

- The `binding`'s child elements are `operations`.
 - These do not define new operations for the port type.
 - Rather, each contains enough information to precisely identify an operation already defined in the port type.
 - These child elements are then combined with extensibility elements to add information specific to a target protocol.



The SOAP Binding

- For SOAP-based services, the WSDL descriptor will include elements specific to SOAP messaging, which **bind** the abstract semantic definitions described thus far to a concrete SOAP implementation.
- Extensibility elements in this binding are:
 - **soap:binding**, which extends the abstract binding by defining the specific SOAP transport (for instance, SOAP over HTTP), and the messaging style as either “document” or “rpc”
 - Other, more specific extensibility elements can also define messaging style, overriding anything defined at a higher level.
 - **soap:operation**, which extends the abstract operation with attributes such as the related SOAP action URI
 - **soap:body**, which can extend an abstract message type by defining “encoded” or “literal” use, encoding rules, and other attributes
 - **soap:fault**, which extends abstract fault messages the same way **soap:body** extends input and output messages
 - **soap:address**, which can be used to assign a real URI to an abstract component such as a service

Binding for Math

EXAMPLE

- Here is a different slice of the Math descriptor, showing the SOAP bindings:

```
<binding name='MathBinding' type='tns:Math'>
  <soap:binding style='rpc'
    transport=
      'http://schemas.xmlsoap.org/soap/http'
  />
  <operation name='Add'>
    <input>
      <soap:body use="encoded"
        namespace="(Math type NS)"
        encodingStyle="(SOAP encoding URI)"
      />
    </input>
    <output>
      <soap:body use="encoded"
        namespace="(Math type NS)"
        encodingStyle="(SOAP encoding URI)"
      />
    </output>
  </operation>
  ...
</binding>
...
<service name='MathService'>
  <port name='Math' binding='tns:MathBinding'>
    <soap:address location=
      'http://localhost:8080/Math/JAX-RPC/Calculate' />
  </port>
</service>
```

Binding for Math

EXAMPLE

- The `soap:binding` specifies SOAP/HTTP and RPC-style messaging.
 - More on document- and RPC-style messaging on the next page.
- The `soap:body` elements specify that SOAP Section 5 encoding will be used for the messages; the namespace URI for the Math service types is provided for encoding purposes.
- Not all the SOAP extensibility elements appear in the WSDL `binding`.
- The `port` element contains the `soap:address` element, which announces to the consumer of the WSDL descriptor that a running service matching this description can be found at the given URI.
 - This element stands somewhat apart from the rest, as it has to do with location, not with semantics.
 - It is possible to consume the rest of the WSDL document and either ignore or override the address by simply connecting elsewhere at runtime.
 - Location information is arguably not the domain of a service descriptor: traditionally this falls to a **naming** service, and in the emerging Web services architecture it will be registered in a repository using UDDI.
 - In either case, the location will be derived dynamically.

Binding for Math

EXAMPLE

- Here is an example of valid message content according to this service descriptor:

```
<soap-env:Envelope xmlns... >
  <soap-env:Body>
    <math:Add>
      <x>6.0</x>
      <y>3</y>
    </math:Add>
  </soap-env:Body>
</soap-env:Envelope>
```

- Note that the **part** names map to element names under a single parent element for the **input** message.
- This parent element's name is governed by the name of the **input, output or fault** component, not the associated message type name.
 - In the absence of a name for this component, the parent **operation** name will be used, with a standard suffix to distinguish requests and solicits from responses.

Document vs. RPC Style

- According to the WSDL specification, SOAP messages can be formed according to either of two dominant practices:
 - The **document** style allows for arbitrary content under the SOAP body element. It is well-suited to passing prepared documents for presentation, or XML fragments.
 - The **RPC** style is best suited to encoding of remote method invocations, such that the body will have a single child element identifying a method to be invoked, and this will have one child for each method parameter, supplying the arguments.
- The practical implication of the choice in the SOAP binding is the presence or absence of an element intervening between the SOAP body element and the message elements that map to WSDL parts.
 - For RPC-style messages this will exist, and parts will be grandchildren of the SOAP body through this element. This is the **add** element in the example message.
 - For document-style messages it will not; the message parts will be direct children of the SOAP body element.
- Message style can be defined at binding, operation or body levels, each level overriding the higher ones.
- Document-style is the default, if no styles are defined at any level to control a particular message.

Encoded vs. Literal Use

- **Information in SOAP messages can be encoded according to either of two standards:**
 - Message content can be controlled directly by a concrete schema, for instance an XML Schema. This is known as **literal** use.
 - Alternatively, message content can be governed by a set of encoding rules, which may incorporate one or more other schema but which set their own standards by which some types are encoded. This is called **encoded** use.
- **The encoded use calls for additional information.**
 - Especially, the encoding rules in play must be identified – this is the meaning of the **encodingStyle** attribute, whose value will be a URI identifying the rule set, such as the SOAP Section 5 encoding.
 - The **namespace** attribute is also relevant to the encoding as it may identify relevant types for message content.
- **The choice of literal vs. encoded usage is orthogonal to the choice of document vs. RPC message style.**
 - RPC/encoded and document/literal are common combinations, however.

The HTTP Binding

- As an example of WSDL's extensibility, we'll survey the HTTP GET/POST binding.
 - This binding can be used to describe interactions with a Web application or service using HTTP GET/POST requests.
 - This binding is more aptly used with REST-based Web services than with those that use SOAP; see the Learning Resources appendix for more on REST.
- HTTP extensibility elements are:
 - **http:binding**, which identifies the HTTP binding itself and indicates which verb, GET or POST, is to be used
 - **http:operation**, which can define a URI relative to the address (below) for each abstract operation
 - **http:address**, which identifies the service location as a URI
 - **http:urlEncoded** and **http:urlReplacement**, which define two mutually-exclusive means of encoding the abstract message parts into the CGI string for GET requests
- The HTTP binding also uses parts of the MIME binding, to define message parts that travel as attachments.
- We will not consider non-SOAP bindings any further in this course.

Dynamic Invocation

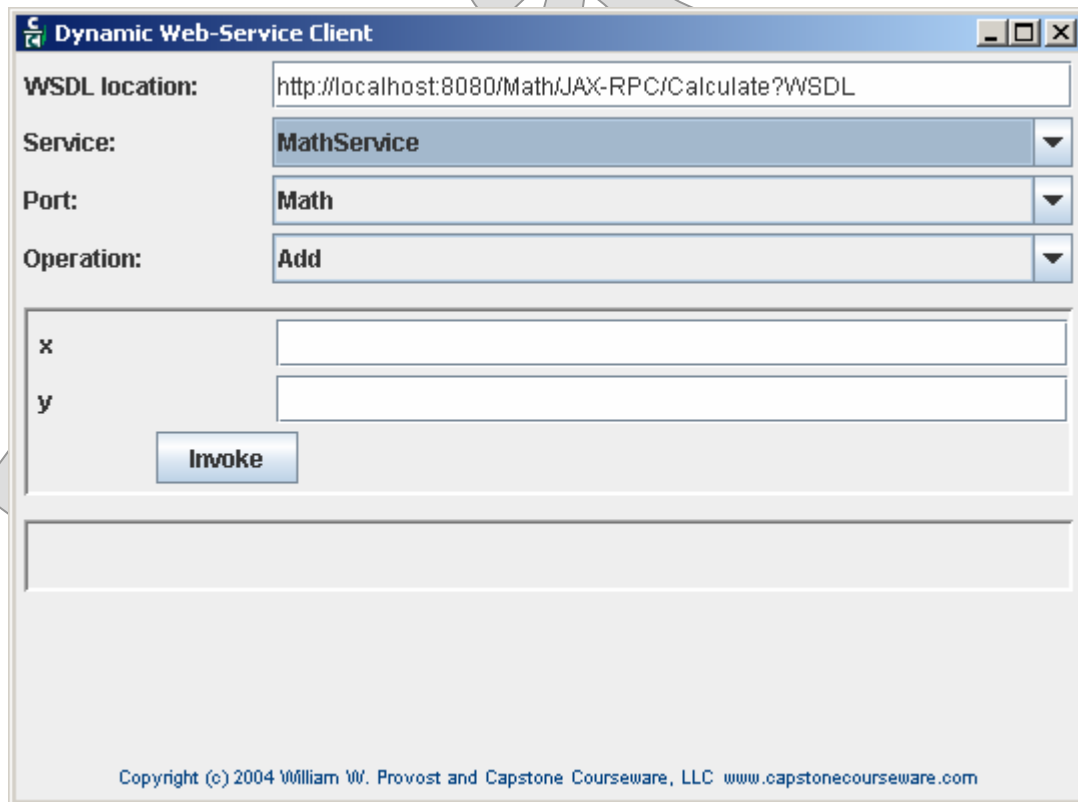
DEMO

- We'll look at a demonstration of dynamic invocation of a Web service based on WSDL descriptors.
- The **Dynamic** application reads a WSDL descriptor and dynamically populates choosers based on the components it finds there.
 - The user can select a service, and per service a port, and per port an operation.
 - The application then populates an area of the GUI with input controls based on the chosen operation's message signature. (The application can only handle input of simple types.)
 - The user can enter argument values and send a message that should be valid according to the WSDL.
 - The response is posted in another dynamically-built GUI.
 - The application uses JAXP to parse the WSDL descriptor as XML and SAAJ to perform SOAP messaging.
- The application is in **Demos\Dynamic**.
- For this demonstration we'll need some running Web services.
 - The Math service can be built from **Examples\Math\JAX-RPC**.
 - The PigLatin service can be built from **Examples\PigLatin\WSDLToJava**.

Dynamic Invocation

DEMO

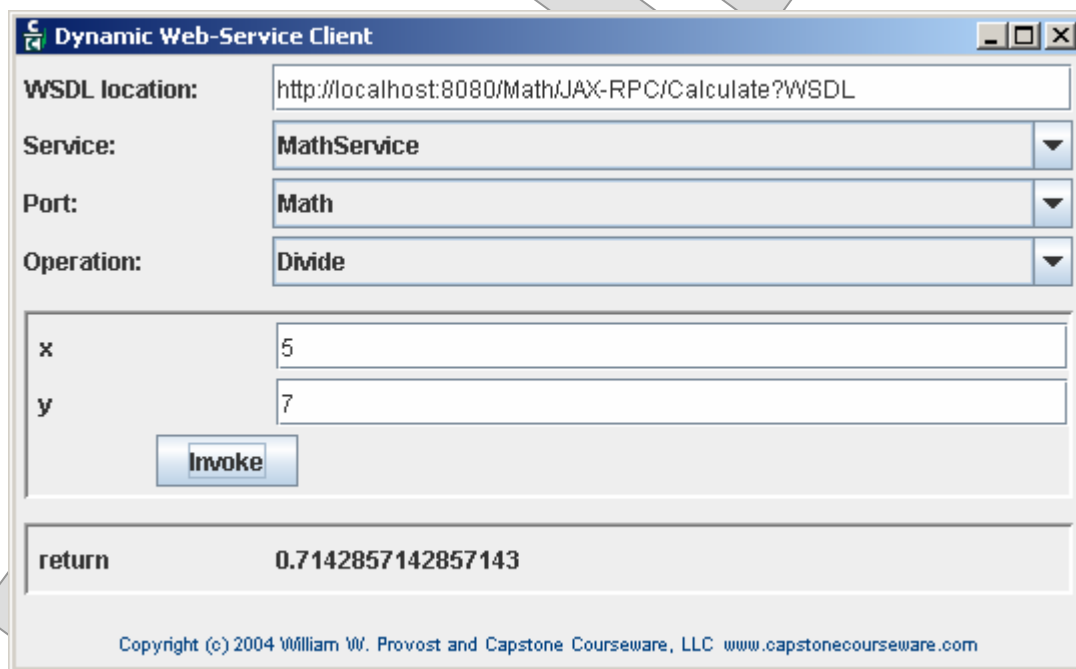
1. Begin by building the two target services: in each project directory, just run **asant**.
2. Build and test the application using the **build** and **run** scripts.
3. Enter the URI for the Math service descriptor, as shown below, and hit the tab key to move to the service chooser. The application will pause while it downloads the WSDL file, and then will populate the combo boxes with information on available services, ports, and operations:



Dynamic Invocation

DEMO

4. Tab through the combo boxes, choosing whichever operation you like in the third one. Whenever you select from this combo box, the GUI will be populated with text fields labeled appropriately for the request message parts – in this case **x** and **y** arguments. You can enter values in these text fields and click the **Invoke** button. When the service has responded, the result will appear in another addition to the GUI:



5. You can also see the raw SOAP messages that were exchanged on your behalf by the application – these are echoed to the console.

Dynamic Invocation

DEMO

6. Now we'll try the PigLatin service. Take a quick look at the sole **portType** descriptor for this service, along with its constituent messages:

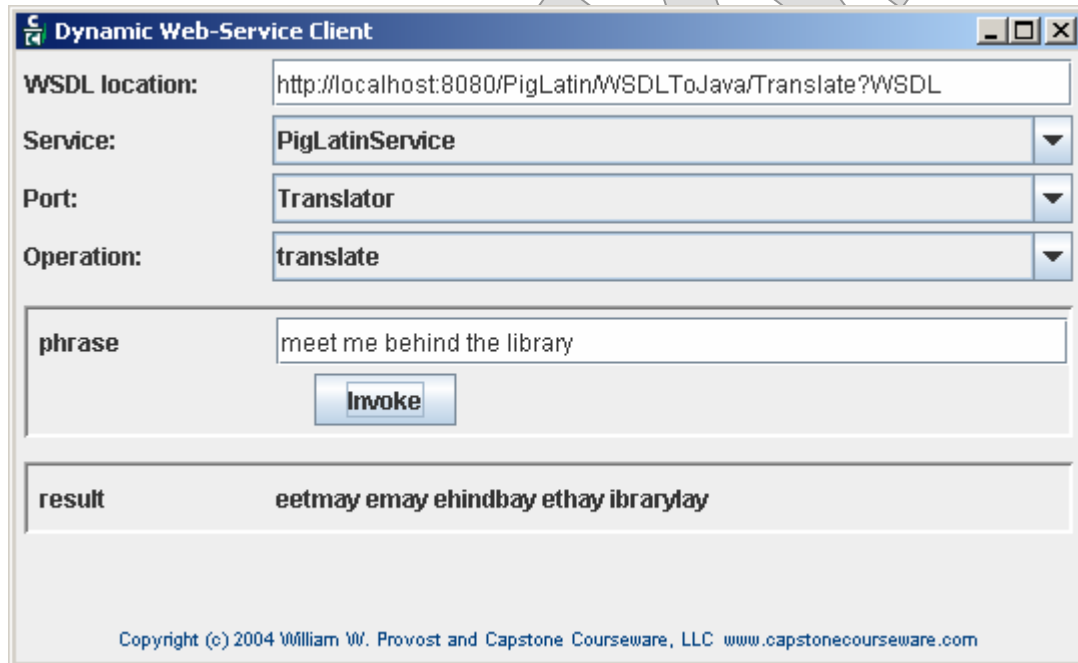
```
<message name='EnglishPhrase'>
  <part name='phrase' type='xsd:string' />
</message>
<message name='PigLatinPhrase'>
  <part name='result' type='xsd:string' />
</message>

<portType name='Translator'>
  <operation name='Translate'
    parameterOrder='phrase'>
    <input message='tns:EnglishPhrase' />
    <output message='tns:PigLatinPhrase' />
  </operation>
</portType>
```

Dynamic Invocation

DEMO

7. Enter the URI for this WSDL file, as shown below, and run through the process again. Here you have no choice of service, port, or operation, but the input and output GUI sections do adjust to the different semantics for the **Translate** operation.



The screenshot shows a Java Swing window titled "Dynamic Web-Service Client". The window contains the following fields and controls:

- WSDL location:** A text field containing the URI `http://localhost:8080/PigLatin/WSDLToJava/Translate?WSDL`.
- Service:** A dropdown menu with "PigLatinService" selected.
- Port:** A dropdown menu with "Translator" selected.
- Operation:** A dropdown menu with "translate" selected.
- phrase:** A text field containing the text "meet me behind the library".
- Invoke:** A button labeled "Invoke" positioned below the phrase field.
- result:** A text field containing the translated text "eetmay emay ehindbay ethay ibraraylay".

At the bottom of the window, there is a copyright notice: "Copyright (c) 2004 William W. Provost and Capstone Courseware, LLC www.capstonecourseware.com".

WSDL for the Order Tracking Service

LAB 7B

In this lab you will complete the WSDL document from Lab 7A by adding concrete information on service, port, and SOAP/HTTP binding. You will test your descriptor by providing it to the **wscompile** tool and using it to generate service stub classes, which can be used by a prepared client application.

Detailed instructions are contained in the Lab 7B write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

Deploying WSDL

- It only does so much good to create a service descriptor, if that descriptor cannot be located by a potential client or partner!
- WSDL documents are typically published on a website alongside the services that they describe.
- A consensus has emerged recently on where to locate a service's WSDL document(s).
 - For any service endpoint URI, the WSDL can be found by appending the suffix “?WSDL”. Thus:

`http://localhost:8080/Math/JAX-RPC/Calculate?WSDL`

- Microsoft's .NET framework got this started, and major vendors on the Java side have jumped on the bandwagon.
- Note that this is only a convention, not a true standard.
- One downside, too, is that attributes of the service endpoint URI – for instance authentication constraints – will be inherited by this WSDL URI, which may or may not be desirable. We'll see this practical impact in a later chapter.
- It is sometimes better to deploy the WSDL at a sibling path such as the following:

`http://localhost:8080/Math/JAX-RPC/WSDL/Math.wsdl`

- WSDL documents are also commonly published to UDDI repositories as business information.

SUMMARY

- **WSDL is the consensus solution to the problem of Web-service description.**
- **As such it solves a number of key problems for SOAP-based Web services:**
 - External to the Web service messages, it defines fundamental type information for purposes of generating and/or validating message content.
 - It models the messages used by a Web service to allow higher-level automation of components in the system, including services themselves, clients, and intermediaries.
 - This automation can come in the form of dynamic, interpretive tools, or of generated code dedicated to particular message types.
- **A WSDL descriptor can be the document of record for a Web service.**
 - This may or may not mean that code is generated from the WSDL, but always means that the WSDL descriptor accurately describes the service's runtime behaviors.
- **Descriptors can be shared between service implementer and client developers, intermediaries and other tools.**
- **They can also be registered with service repositories, via such protocols as UDDI.**

Abstract WSDL for Order Tracking

LAB 7A

Introduction

In this lab you will build the abstract WSDL model of messages supported by the Order Tracking service. For now, you will validate your descriptor by passing it through the **wscmpile** tool; in a later lab you will use the complete WSDL descriptor to generate service stub classes, which can be used by a prepared client application.

Suggested Time: 30 minutes.

Directories:

- Labs\Lab07A** (do your work here)
- Examples\Tracking\Step0\Client** (backup copy of starter code)
- Examples\Tracking\Step1\Client** (contains lab solution)

Files: WSDL\OrderTracking.wsdl

Instructions

1. Open the starter file **OrderTracking.wsdl**. The **types** section has already been written, and its schema includes the simple and complex domain types called for by the UML design shown earlier. (The **Contents** type is not explicitly called out in the UML design, but is necessary as a means of establishing the one-to-many relationship in the design as a SOAP Section 5 array.) The **Tracking** service interface has not yet been defined, however.
2. Analyze the **Tracking** interface as depicted below and decide what SOAP message types you will need to describe. Note that each method on the interface will become an **operation**, and each operation will require a request message type and a response message type (that is, an **input** and an **output** message). So, you will define six message types at a maximum; remember, too, that message types can be reused by multiple operations. How many distinct message types do you see?

Tracking
<pre> checkStatus(ID : long) : StatusEnum getDetail(ID : long) : Order cancel(ID : long) : StatusEnum </pre>

3. Clearly, since all three methods take a single **long** parameter, we will need a message type to carry this information. Define a **message** element with the name "OrderID". Give it a single **part** element as a child, with the name "ID" and the type **xs:long**.

4. The **checkStatus** method returns a status indicator; we will need a response message type to carry this information. Right after the **types** section, define a message called “Status” with a single part called “Status” whose type is **order:StatusEnum**. Note that this message type will work as the response type for the **cancel** method as well.
5. Finally, the **getDetail** method returns a whole **Order**. Define a message called “Order” with a single part called “Order” whose type is **order:Order**.
6. Save the file and try validating at this point. First, create a directory **Build**, into which code will be generated.
7. Run the **wscmpile** tool to validate the descriptor content as shown below. You will see a warning, as shown, and no code will be generated, but most errors in structure or syntax of your WSDL components will be caught and reported. If not, investigate any errors and double-check your **message** elements.

```
wscmpile -gen:client -keep -d Build config.xml  
warning: WSDL document does not define any services
```

8. Now we have the fundamental message types defined; let’s move up a level and define the **Tracking** interface as a whole. This means creating a **portType** component, with the name “Tracking”.
9. For each of the three methods, define an **operation** element as a child of the **portType**. Give it the name of the corresponding method, and give it child elements **input** and **output**, each with a **message** attribute identifying the correct in or out message type. (Remember to provide the proper namespace prefix for each of the message names; the target namespace for the WSDL descriptor is **order:.**)
10. Compile again, and fix any errors reported, until the tool runs cleanly. Compare your completed WSDL to the answer version.

WSDL for the Order Tracking Service

LAB 7B

Introduction

In this lab you will complete the WSDL document from Lab 7A by adding concrete information on service, port, and SOAP/HTTP binding. You will test your descriptor by providing it to the **wscmpile** tool and using it to generate service stub classes, which can be used by a prepared client application.

The service is already implemented for this lab, although in a later lab you will build it yourself from scratch. The pre-built service already has a WSDL file, and in the real world this file would simply be transmitted to the client developer. This lab is meant to provide exercise in writing WSDL by hand and a deeper knowledge of the various WSDL components: to these ends you will create a separate WSDL descriptor just for the client. No peeking!

Suggested Time: 30 minutes.

Directories:

Labs\Lab07B	(do your work here)
Examples\Tracking\Step1\Client	(backup copy of starter code)
Examples\Tracking\Step2\Client	(contains lab solution)
Examples\Tracking\Step2\Service	(target service)

Files:

WSDL\OrderTracking.wsdl
cc\biz\Tracker.java

Packages: **cc.biz**

Instructions

1. Build and deploy the prepared Order Tracking service: set a console working directory to **Examples\Tracking\Step2\Service**, and simply run **asant**.
2. Now, in the lab directory itself, open the starter file **OrderTracking.wsdl**. This is the completed version from Lab 7A, and so includes the **types** and the abstract model: **message**, **operation**, and **portType** components.
3. Now you must describe the concrete service model using SOAP over HTTP. This means creating a **binding** component; call it “TrackingBinding”.

4. The binding's **type** attribute identifies the port type that's being bound to a specific protocol implementation; set this to **order:Tracking**. (Note again the use of the namespace prefix to refer to a previously-defined type. However, note also that the child **operation** elements you're about to define will not need this prefix. The idea is that the binding has already qualified description to a certain port type, and so the operations within this binding can be named locally and reference child **operations** of that port type.)
5. As the first child of the binding element, add a **soap:binding** element (the **soap:** namespace prefix is already defined). This identifies the parent as a SOAP binding, and provides some general information about the messaging model. Give this element a **transport** attribute that identifies the HTTP transport by the URI shown below. Set the attribute **style** to "rpc".

```
transport="http://schemas.xmlsoap.org/soap/http"
```

6. As the second child of the binding element, add an **operation** with the name "checkStatus". Give this an **input** and an **output** as child elements, and for each of these declare a single child element of the name **soap:body**. The body will define **encodingStyle** to be the SOAP Section 5 encoding (use the URI shown below); **use** as "encoded", and the **namespace** for type information – copy this URI from the **targetNamespace** for the WSDL descriptor itself, found at the top of the document.

```
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

7. Create **operation** elements for each of the other two interface methods **getDetail** and **cancel**; other than the operation name, the content for each of these will be identical to the one you've just created for **checkStatus**. This completes the binding descriptor; you now have a concrete SOAP/HTTP messaging model described.
8. Lastly the service itself must be identified: create a **service** element with the name "TrackingService". Its one child element will be a **port** element with the name "Tracking" and a **binding** attribute identifying **order:TrackingBinding**.
9. As a child of this **port** element, create a **soap:address** element, with a **location** attribute as shown:

```
location="http://LOCALHOST:8080/Tracking/Tracker"
```

10. Now we will try to get this descriptor into use. Take a quick look at the pre-built client code in **Tracker.java**, and note that it connects to the service through a **Tracking_Stub** class. This stub implements the generated Web service interface so that it has one Java method for each **operation** on the **portType**:

```
cc.biz.Tracking_Stub service = (cc.biz.Tracking_Stub)
    new cc.biz.TrackingService_Impl ().getTracking ();
...
if (command.equalsIgnoreCase ("check"))
    System.out.println (service.checkStatus (ID));
```

11. Run **wscompile** as instructed in Lab 7A; fix any validity errors before proceeding.
12. Take a look at the code that's been generated under the **Build** directory. Note that there is a Java class for each of the types in the **types** section of the descriptor; a request and response "struct" for each operation; an interface for the port type; and a class for the service. We'll look much more closely at the roles these interfaces and classes play in the JAX-RPC architecture in the following chapters.
13. Build the client by running **asant**. This will repeat the **wscompile** step and will also compile the dependent Java client classes.
14. Run the client using the provided **Track** script. Pass the command **check** and the order number **2** as arguments to this script. You should see output similar to that shown below, indicating that the service was successfully located and invoked.

Track check 2
Back-ordered

Optional Steps

15. Start the **SOAPSniffer** by running **Examples\SOAPSniffer\Sniff**. Run the client again, passing an altered URL that includes port 8079 instead of 8080, and observe the mappings from WSDL types, parts, messages, and operations to actual SOAP message content. (Messages are reproduced following the lab answer code.)

Track check 2 `http://localhost:8079/Tracking/Tracker`

16. Try running the **Dynamic** application as in the demo, pointing it at the URI shown below. You should see that the WSDL is correctly interpreted to build a GUI that asks for the order ID, as shown below. Invoking any of the methods will fail, however: each method returns a type that is more complex than this application was built to be able to figure out, so there will be an exception.

`http://localhost:8080/Tracking/Tracker?WSDL`

