



CHAPTER 2
**WEB SERVICES DESCRIPTION
LANGUAGE**



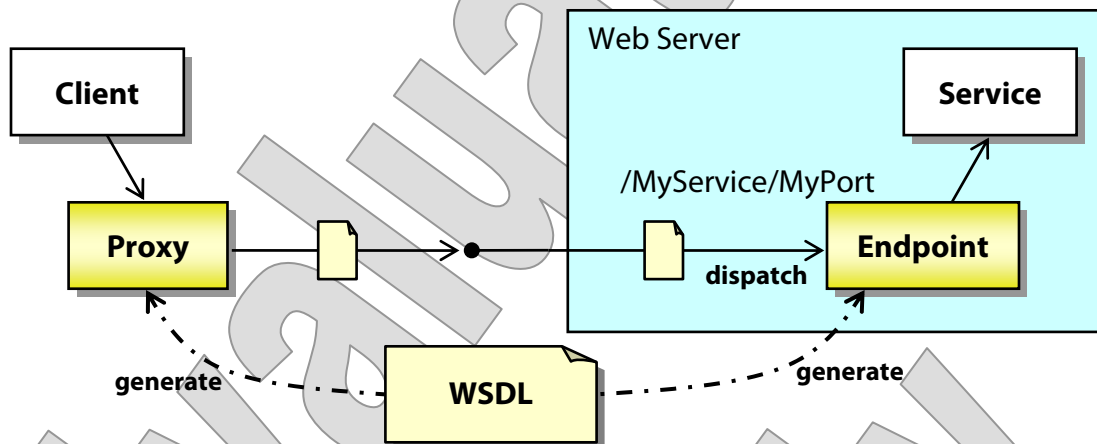
OBJECTIVES

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

- Explain the importance of providing full metadata for a web service, and how WSDL accomplishes this.
- Describe the WSDL model for creating one’s own
 - Abstract service models
 - Concrete service models
- Write WSDL documents to describe message and service types.
- Create bindings for a service description to a specific protocol, such as SOAP.
- Generate client code for a WSDL-described service.

Metadata for Web Services

- As we discussed earlier, web services do not live by SOAP alone.
- Service **description** is a critical element of the architecture; there are several key benefits:
 - Developers can **understand the contract** in which their software will play a part – service, client, or intermediary.
 - One can **generate code** for either service or client development that handles the grunt-work of SOAP messaging:



- It's possible to **validate** message content at runtime – though this is not necessarily something we'd want to do for each request and response, for performance reasons.
- Metadata enables **dynamic invocation** and reception of messages, by general-purpose intermediaries or possibly as part of a rapid-application-development tool that helps users build composite applications from a set of available services.

Web Services Description Language

- Broad consensus has formed around the **Web Services Description Language**, or **WSDL**, as the best means of providing the metadata necessary to web services and SOAs.
- WSDL is an XML vocabulary for describing web services.
- A WSDL descriptor defines:
 - **Message types** and their valid content
 - **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
 - Service **interfaces**, essentially compositions of operations
 - Service **implementations** – or perhaps “defines” is the wrong verb for this one; let’s say WSDL “identifies” service implementations, which naturally will be written in some other language
 - **Bindings** between interfaces and implementations – in fact the choice of SOAP/HTTP is considered part of this binding, as the interface is not protocol-specific

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/`

- Remember the warning from the SOAP chapter about the importance of that trailing slash ...

- Each **binding** of WSDL to a possible messaging protocol is given its own namespace:

- The SOAP binding namespaces, commonly prefixed **soap:**

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

`http://schemas.xmlsoap.org/wsdl/soap12/`

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

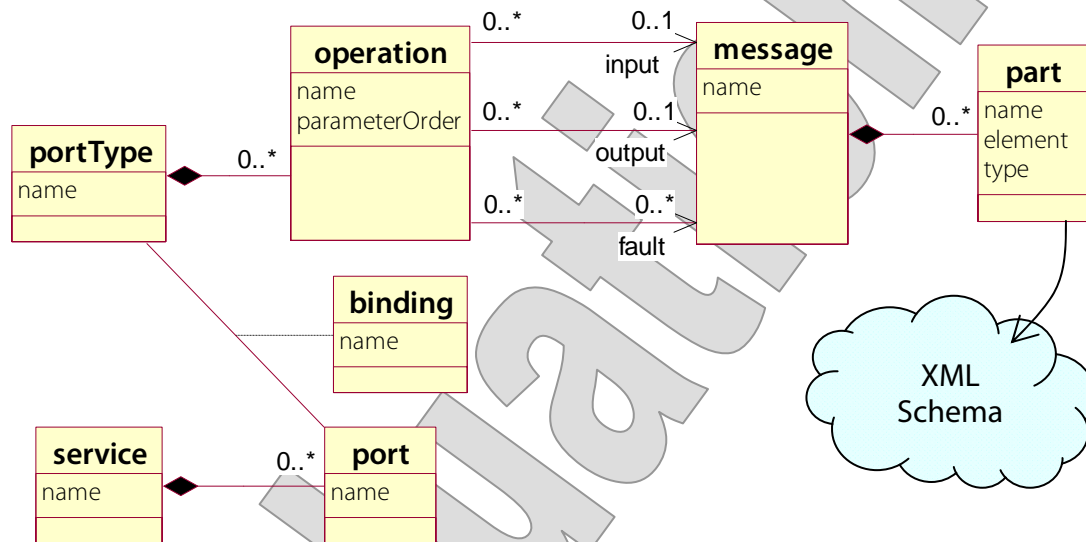
`http://schemas.xmlsoap.org/wsdl/http/`

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

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

The Description Model

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



- There are components to define an **abstract model** of behavior:
 - A `<portType>` is a named interface to a set of `<operation>`s.
 - Each `<operation>` describes a messaging scenario in terms of `<input>`, `<output>`, and `<fault>` messages.
 - Those `<message>` definitions comprise `<part>`s, which in turn use XML Schema global element and type definitions to fully explain the model of expected message content.
- Then other components can define the **concrete model**:
 - The `<service>` component identifies a web service proper.
 - A service as envisioned by WSDL can offer multiple `<port>`s (though in practice almost all services are single-port).
 - A `<port>` is guaranteed to implement a given `<portType>`, in a specific way that is described by a `<binding>`.

A WSDL Document

EXAMPLE

- Here is a simple WSDL document – find this in **Examples/Math/JAX-WS/Service/docroot/WEB-INF/wsdl/Math.wsdl**:

```
<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="Calculator">
    <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="Math">
    <port name="Calculator"
      binding="tns: CalculatorBinding">
      <soap:address
        location="SERVER_WILL_COMPLETE"
      />
    </port>
  </service>

</definitions>
```

WSDL and XML Schema

- A WSDL descriptor is not a W3C XML Schema document.
- But it has a close relationship with XML Schema.
 - A WSDL can embed an XML Schema, using its `<types>` component, and elements and types defined in that schema can be used in message `<part>`s.
 - A schema so embedded may also **import** other schema.
- WSDL also follows a number of patterns from XML Schema, and really is of a similar nature.
 - Both are **metamodels**: i.e. models for defining other models.
 - Both can be said to **populate a namespace** with type information: WSDL models govern web-service messaging interactions, and XML Schema models govern the XML content of a specific document or message.
 - Both WSDL's `<definitions>` and Schema's `<schema>` components have **targetNamespace** attributes

Associations Between Components

- Another XML Schema pattern followed by WSDL is the use of both **composition** and **association** to connect components.
 - A `<portType>` is **composed** of `<operation>`s: it has direct child elements that represent the operations it offers.
 - An `<operation>` is **associated** with a `<message>`: it identifies the message by name and that name is matched to the unique name of a message component.
- You may notice something odd about the way names are matched, though – refer back to **Math.wsdl** to see this:
 - The **message** attribute used by an operation's `<input>` component to identify the message will include a namespace prefix.
 - The corresponding `<message>`'s **name** attribute will not.
- This is the rationale:
 - When **defining** a component, we're implicitly populating the WSDL's **target namespace**.
 - When **identifying** another component for association, however, there is **no implicit namespace** – nor should there be.
 - One WSDL must populate one namespace, but it might refer to components from many namespaces.
 - This is why, even though it may seem redundant in many cases, attributes that seek to identify other components must be **qualified names**.

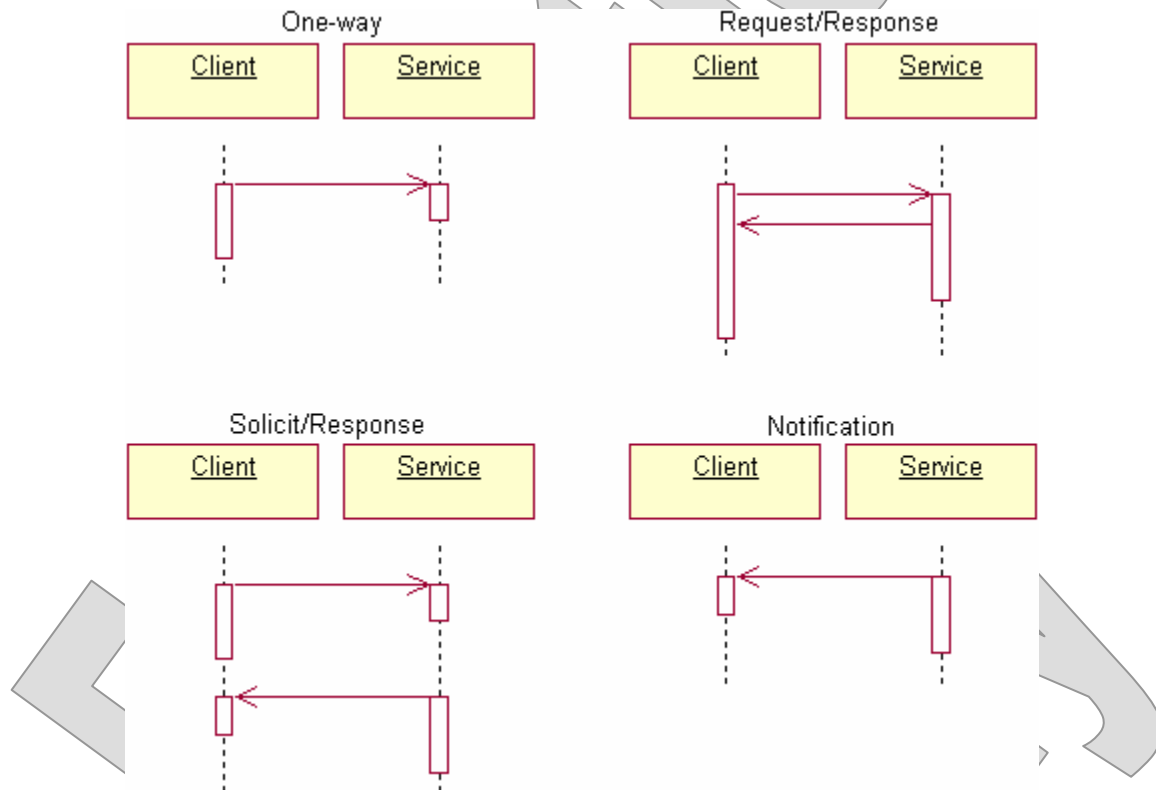
The portType and operation Components

- A `<portType>` includes any number of `<operation>`s, and it has a name – that’s really the whole content model.
- `<operation>` is where the interesting stuff happens; it holds:
 - An optional `<input>` message
 - An optional `<output>` message
 - Zero-to-many `<fault>` messages
- As `<portType>` indicates an interface, `<operation>` is a method on that interface.
- Each input, output, or fault component identifies a `<message>`.
 - Each can also carry its own unique **name**, but this is rarely done for input and output since they are unique by type.
 - Faults are almost always named, even if there’s only one.
- Consider a hypothetical method in a programming language:

```
double calculateMedian (double[] values);
```
- “calculateMedian” would be the name of a corresponding WSDL operation, and it would have input and output messages.
 - One or more faults would correspond to exception signatures.
- `<operation>` also has a `parameterOrder` attribute: this is a hint to processors – especially code generators – as to how the multiple parts of input and output messages might be arranged into a method signature.
 - We’ll consider this again when we look at mapping WSDL to Java.

Operation Modes

- You've noticed that both input and output messages are optional.
- WSDL defines four **operation modes**, each of which indicates a different combination of input and output messages and different synchronization characteristics between them.



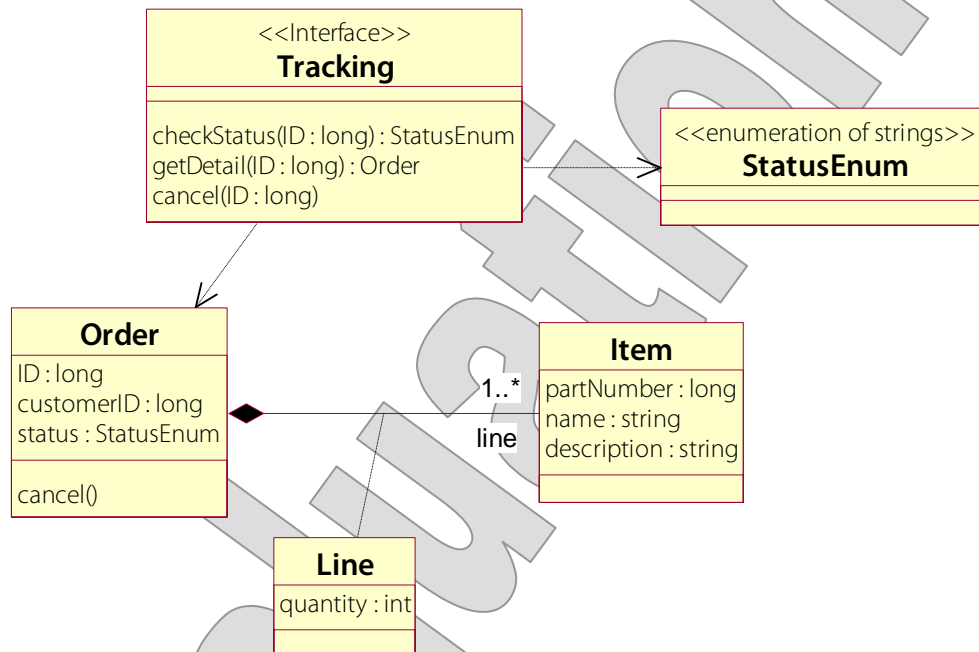
- The latter two are not intuitive, and they imply that the “client” can receive requests and is somehow registered with or known to the service.
- These two are really second-class citizens in the WSDL specification.
 - Bindings are only defined for **request/response** and **one-way**, and that’s where most tools put their effort as well.

The message and part Components

- The `<message>` element defines the exact information that's expected in a single message – whatever direction that message might travel, whatever it might be expected to mean.
 - Notice that inputs and outputs on multiple `<operation>`s can (re)use a given `<message>` definition.
- A `<message>` may have zero to many `<part>`s.
 - The total message content is understood to include the content models of each of the parts, in sequence.
- Each `<part>` has a **name** and must define its content in one of two ways:
 - Using a **type** attribute to identify an XML Schema complex type – in this case the part **name** may appear in the actual message
 - Using an **element** attribute to identify a global element definition – in this case we're identifying an element with its own name and type, so the part's **name** is irrelevant to the message content
- The first of these uses is appropriate to RPC-style operations, and the second is correct for document-style operations.

The Order-Tracking Service

- We'll look now at a new case study: an application that tracks information about product orders:



- **The Tracking interface acts as a façade over a domain model:**
 - **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.
- **In upcoming labs you'll build various pieces of this service – starting in this chapter, with its WSDL descriptor.**

Validating WSDL

- For this chapter's labs we need a WSDL-to-Java code generator.
- For this purpose we'll use the **wsimport** tool that's part of our JAX-WS implementation.
- At times during these labs, you'll run a script **validate.bat** that invokes **wsimport**:

```
wsimport -d build/classes %1
```

- This will result in some code generation as a by-product.
- Eclipse provides a fairly nice GUI-driven WSDL editor, and you will get some built-in validation using this tool.
 - Here for example is the **Search** port type for the Housing service:

Search		
searchForHousing		
input	parameters	searchForHousing
output	parameters	searchForHousingResponse
getSummaries		
input	parameters	getSummaries
output	parameters	getSummariesResponse
getDetail		
input	getDetail	getDetail
output	getDetailResponse	getDetailResponse

- For the upcoming lab at least, stick to editing in the text view, so we can get a better feel for the nuts and bolts of the language.

An Abstract Model for Order Tracking

LAB 2A

Suggested time: 30 minutes

In this lab you will build the abstract WSDL model of messages for the Tracking service. For now, you will simply validate your descriptor; in a later lab you will use your completed descriptor to generate proxy classes which can be used by a prepared client application to invoke a prepared web service.

Detailed instructions are found at the end of the chapter.

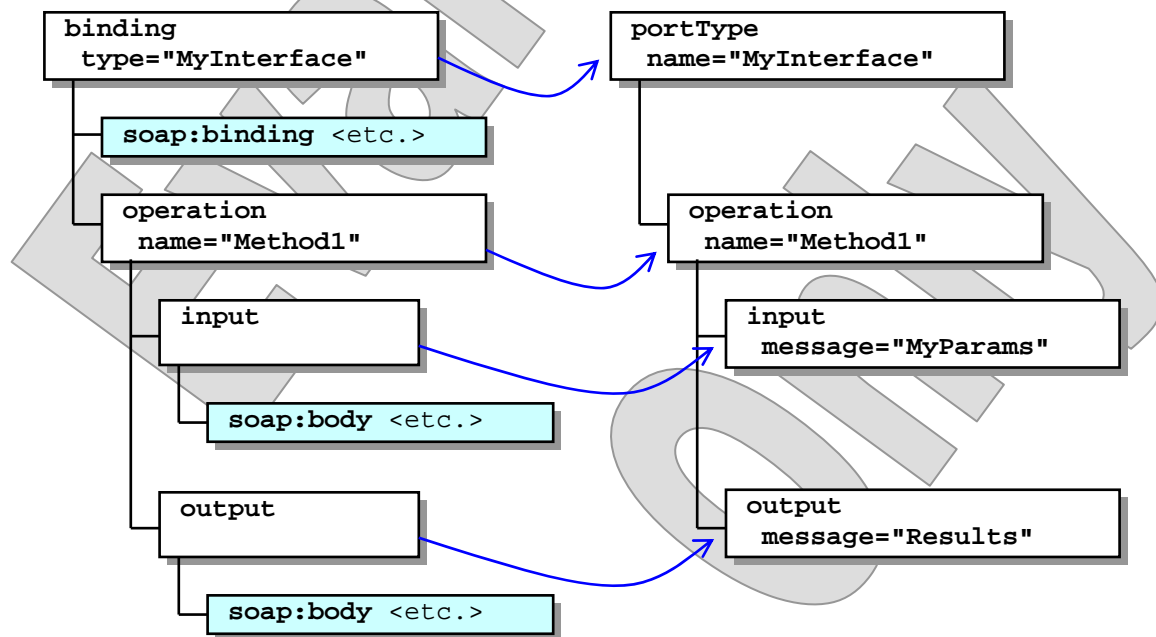
Evaluation Only

Making it Real

- These abstract models are nice, but when do we get to define an actual running service?
- We'll look at the concrete-model components now, starting with `<service>`, which has a name and a collection of `<port>`s.
- A `<port>` promises to implement a `<portType>`, which is identified by way of a `<binding>`.
- `<port>` and `<binding>` are both in the WSDL namespace – they are standard WSDL components – but they usually rely on **extensibility elements** to say what they really need to say.
- The WSDL vocabulary allows the inclusion of elements from non-WSDL namespaces at various points in a descriptor.
 - These are called **extensibility elements**.
- 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.
 - This leads to names such as **soap:binding** that, at first blush, may seem to be from the SOAP envelope namespace itself.
 - Remember that namespace prefixes can be assigned at the author's choice to any namespace URIs; SOAP is just an awfully common token in web services.

The binding Component

- One WSDL component, the `<binding>`, exists just for the purpose of extending the abstract service model by playing host to various extensibility elements.
 - A `<port>` refers to a `<binding>`.
 - That `<binding>`, in turn, refers to a `<portType>`.
- The `<binding>`'s children are `<operation>`s – but these do not define new operations on the referenced 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” (the default) or “rpc”
 - `<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, namespaces, 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
- Remember that the WS-I BP requires
 - Only literal use, meaning schema rather than encoding schemes
 - Document or RPC style is fine, but it must be consistent over the whole port/type – no mixing and matching at the operation level

Binding for the Math Model

EXAMPLE

- Here's the rest of the **Math.wsdl** from our earlier example:

```
<binding name="CalculatorBinding"
        type="tns:Calculator">
  <soap:binding
    style="rpc"
    transport=
      "http://schemas.xmlsoap.org/soap/http"
  />
  <operation name="Add">
    <input>
      <soap:body
        use="literal"
        namespace=".../Math"
      />
    </input>
    <output>
      <soap:body
        use="literal"
        namespace=".../Math"
      />
    </output>
  </operation>
  ...
```

- The binding reifies the abstract model as follows:
 - It identifies **SOAP over HTTP** as the messaging transport.
 - It says that all operations will be **RPC-style** – we'll talk about the practical impact of this in a moment.
 - It identifies the **namespace** in which the operation element is understood to exist.

Document vs. RPC Style

- We've discussed the differences between document-style and RPC-style services already, but more in the abstract: you know it when you see it.
- In WSDL's SOAP binding, each style implies different arrangement of elements in the actual SOAP message – this is an entirely practical matter.
- In RPC style, the conformant SOAP message will have a single body element which represents the operation itself.
 - The element name will be that of the WSDL `<operation>`.
 - The namespace is defined by the corresponding `<soap:body>` element's `namespace` attribute.
- So, the conformant SOAP request for the **Add** operation as defined in the **Math.wsdl** would look like this:

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

- In a document-style binding of the same abstract model, the `<math:Add>` element would disappear, and the `<x>` and `<y>` elements would be direct children of the SOAP body.
 - Or we'd re-express this same model, using a different message type supported by an XML schema complex type – we'll try this later.

The HTTP Binding

- **Strictly 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.
 - It is more aptly used with REST-based web services than with those that use SOAP.
- **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

EXAMPLE

- In the next few chapters we'll be looking at JAX-WS, which of course relies on a WSDL compiler to generate Java code.
 - That code is then **statically bound** to a specific WSDL model.
- WSDL also enables dynamic binding, and as a rough illustration of this capability we'll now try out a dynamic web-services client application, found in **Examples/Dynamic**.
- This 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 single-value types.)
 - The user can enter argument values and **send a message** that should be valid according to the WSDL.
 - The **response** is interpreted into another dynamically-built GUI.
- The application is actually rather low-tech: it uses JAXP to parse the WSDL descriptor as XML, and SAAJ to perform SOAP messaging.
 - But it gives an idea what more ambitious APIs such as JAXB and JAX-WS can do (and JAX-WS does have some dynamic invocation capabilities).

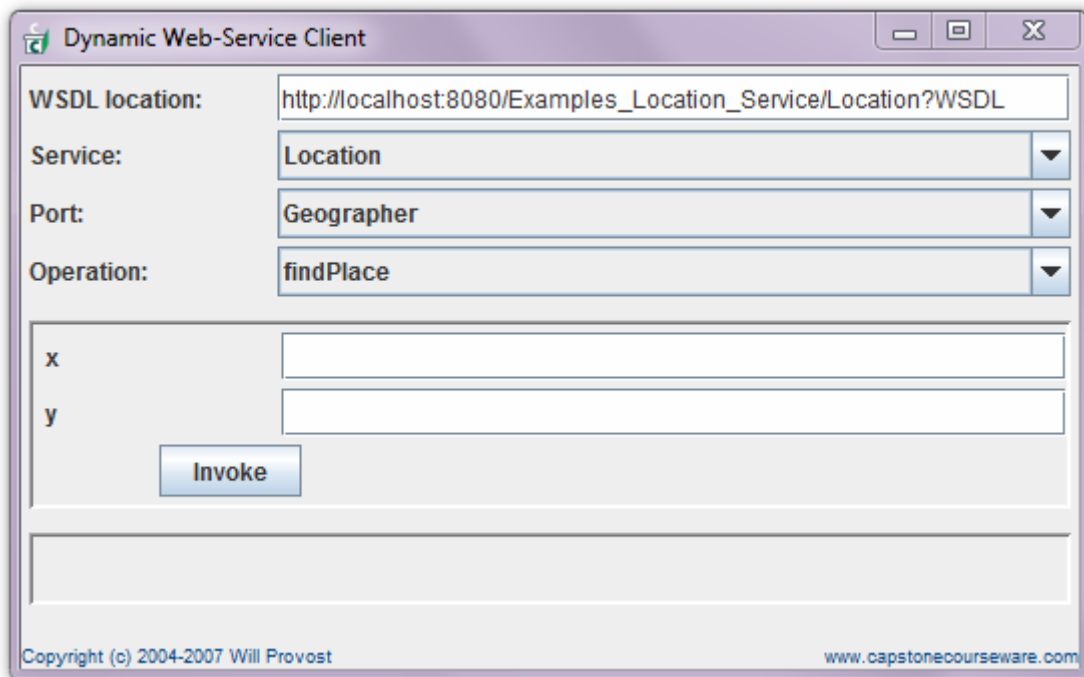
Dynamic Invocation

EXAMPLE

- Build and deploy the service in **Examples/Location/Service**.
- Build the dynamic client application and use the **run** script (or **F10** to run **cc.soap.DynamicWSClient** as a Java application).
- Enter the URI for the Location service descriptor, and hit TAB.

`http://localhost:8080/Location/Location?WSDL`

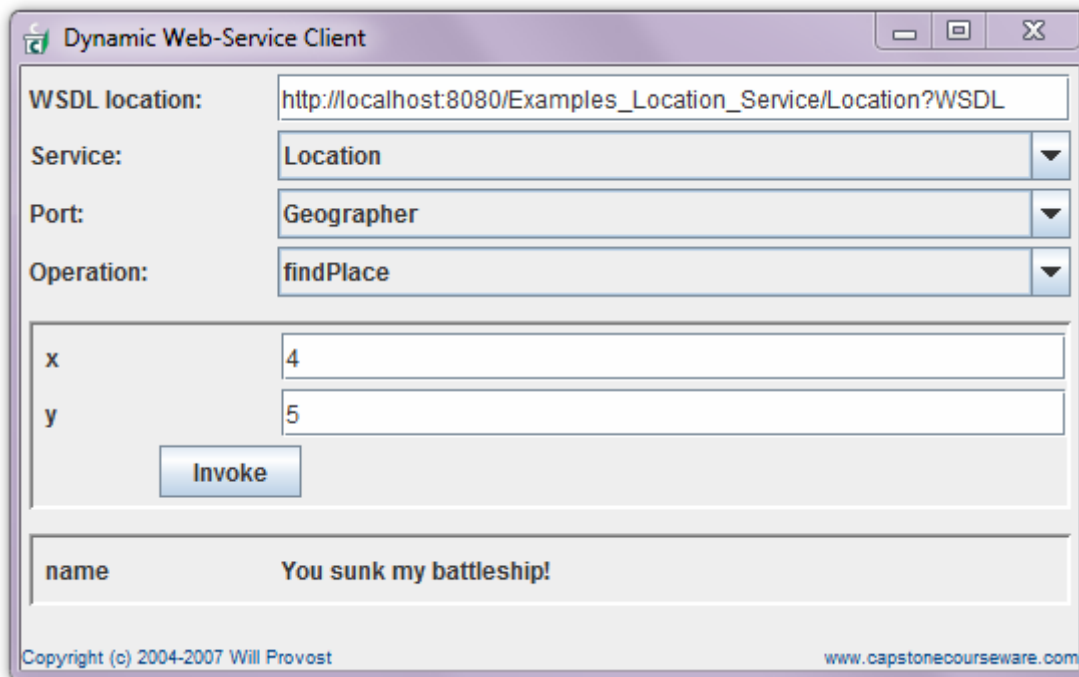
- Or, if you published from Eclipse, replace the first “Location” with the full project name “Examples_Location_Service”, as shown below.
- The application will download the WSDL, and then populate the combo boxes with available services, ports, and operations:



Dynamic Invocation

EXAMPLE

- Tab through the combo boxes to reach the input fields, and enter numeric data.
- Click the **Invoke** button.
 - When the service has responded, the (rather whimsical) result will appear in another addition to the GUI:

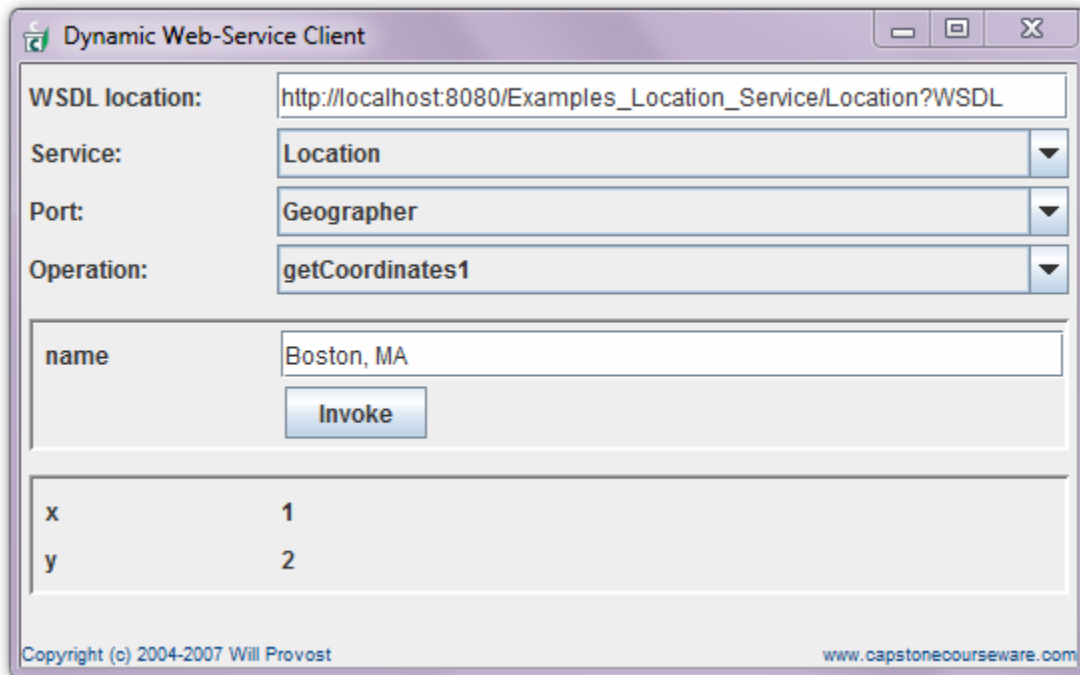


- You can also see the raw SOAP messages that were exchanged on your behalf by the application – these are echoed to the console.

Dynamic Invocation

EXAMPLE

- Make a different selection in the **Operation** box, and see that the client rebuilds the rest of the GUI according to the WSDL-described operation semantics.
 - Here's the result of an invocation to **getCoordinates1**:



- We'll be able to try out other services using this client – including the Tracking service used in the upcoming lab – but we'll also see that this client has several limitations:
 - It can't understand document-style service models, at least not well enough to give you a useful GUI to the services.
 - It can't represent complex-type parameters or response values.

A Concrete Model for Order Tracking

LAB 2B

Suggested time: 30 minutes

In this lab you will complete the WSDL document from Lab 2A by adding concrete information on service, port, and SOAP/HTTP binding. You will again validate your descriptor, and then you will use it in a full Ant build of the client application, using it to generate proxy classes on which a pre-written client application will depend for its SOAP messaging.

Detailed instructions are found at the end of the chapter.

Evaluation Only

Publishing WSDL Descriptors

- How can potential clients find our service?
 - A WSDL is essential, but then where can clients find that?
- There are a few options, from simple to SOA-ambitious.
- WSDL documents are often 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): simply add “?WSDL” to the service URI.

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

- Application servers implement this automatically – and they fix up the **location** attribute of the SOAP address component while they're at it, so that generated client proxies will find the service without help from the hand-written client software.
- There's a bit of a **chicken-and-egg problem** here, though: how would a client know the service URI, unless they'd see the WSDL?
- This address will also fall under any **security constraints** that you set on the service URI itself – you can't force authentication and authorization over the service endpoint without effectively hiding the WSDL, too.
- So it is sometimes good to deploy the WSDL in duplicate, at a sibling path such as the following:

`http://localhost:8080/Tracking/WSDL/Track.wsdl`

- In proper SOAs, WSDL addresses will also be published in UDDI repositories, naming services, and the like.

SUMMARY

- **WSDL is the consensus solution to the problem of web-service description.**
 - It relies on XML Schema to express precise models for the content of individual messages.
 - It adds its own type information describing the service as a whole, including messaging scenarios (operations) and protocol specifics (bindings) that are gathered into service descriptions.
- **A WSDL description often becomes the “document of record” for a web service or system of services and client software.**
- **Again, it defines a contract, and this allows multiple parties to carry out their part of a larger business collaboration with confidence that**
 - The other roles are being filled
 - All parties will be able to interoperate when they need to
- **It’s a business question, rather than a technical question, who writes or owns the WSDL and who simply uses it.**
 - Several scenarios are possible, and we’ll be visiting many of them in the upcoming chapters.

An Abstract Model for Order Tracking

LAB 2A

In this lab you will build the abstract WSDL model of messages for the Tracking service. For now, you will simply validate your descriptor; in a later lab you will use your completed descriptor to generate proxy classes which can be used by a prepared client application to invoke a prepared web service.

Lab workspace: Labs/Lab02A
Backup of starter code: Examples/Tracking/Step1/Client
Answer folder(s): Examples/Tracking/Step2/Client
Files: WSDL/OrderTracking.wsdl
WSDL/OrderTracking.xsd
WSDL/Fake.wsdl

Instructions:

1. Open the starter file **OrderTracking.wsdl**. The `<types>` section has already been written, and its schema imports the external document **OrderTracking.xsd** that defines the simple and complex domain types shown earlier. The **Tracking** service interface has not yet been defined, however.
2. Start by attempting to validate the starter file, by running the **validate** script as shown below – or, if you’re working from Eclipse, just double-click **validateFromIDE.bat**. (The **Fake.wsdl** file is there to fill in some necessary concrete-model information; otherwise **wsimport** will refuse to process your abstract model. We’ll ditch this file in favor of your own concrete model in the next lab.)

```
validate WSDL/Fake.wsdl
[ERROR] invalid entity name: "Tracking" (in namespace
"http://ebiz.org/OrderTracking/wsdl")
line 0 of unknown location
```

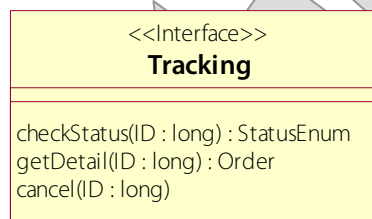
Since we have almost no content, this is not surprising. **wsimport** is trying to work “outside-in” from the `<service>`, `<port>`, and `<binding>` in **Fake.wsdl**, through the `<portType>` and `<message>` types in **OrderTracking.wsdl**, to the schema types that they reference. Ah – but the whole middle’s missing there! and that’s what makes **wsimport** complain: there’s no port type called “Tracking” for it to parse.

Often in WSDL authoring, people work from the inside, out: build schema types, then message types, then operations and port types, and so on. Since **wsimport** likes to work the other way, we’ll do so as well, and start with our port type, and fill in message types after that.

An Abstract Model for Order Tracking

LAB 2A

3. First, though, let's know what message types we'll need, at least to the extent that we can name and count them. 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?



Here's what we'll need:

- Clearly, since all three methods take a single **long** parameter, we will need a message type to carry this information. We'll call this an "OrderID".
 - The **checkStatus** method returns a status indicator; we will need a response message type to carry this information. This will be "Status".
 - Finally, the **getDetail** method returns a whole order object. So we'll have a third message type called "Order".
4. In **OrderTracking.wsdl**, create a **<portType>** component, with the name "Tracking".
5. 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:**.)

Give the **cancel** operation an input message, but no output message. The UML above suggests a one-way operation, and that's what our service ultimately will expect.

An Abstract Model for Order Tracking

LAB 2A

6. Validate now, and see that **wsimport** has found your port type (no more complaints about “Tracking” being an invalid entity name). But (unless there are other problems with the port type you just added) you’ll start to see new errors regarding the **message** attributes that point (so far) to nowhere:

```
validate WSDL/Fake.wsdl
warning: invalid entity name: "OrderID"
(in namespace: "http://ebiz.org/OrderTracking/wsdl")
```

7. 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**. You can put this anywhere, by the way, as long as it’s a top-level element (a child of the root **<definitions>** element); that is, before or after the **<portType>** doesn’t matter.
8. If you validate now, you’ll see a new error – **wsimport** is happy with your first message type but now wants to see the “Status” message type.
9. Define a **<message>** called “Status” with a single part called “Status” whose type is **order:StatusEnum**.
10. Finally, the **getDetail** method returns a whole **Order**. Define a **<message>** called “Order” with a single part called “Order” whose type is **order:Order**.
11. Validate again ...

```
validate WSDL/Fake.wsdl
parsing WSDL...
generating code...
compiling code...
```

... and no news is good news! Fix any errors that are reported, until the tool runs cleanly. When it does, you know that **wsimport** has found the fake service definition in **Fake.wsdl**; parsed it, which leads it to your WSDL file; processed your port type and messages successfully; and in fact read into the imported Schema file as well.

When it does parse the WSDL successfully, **wsimport** also generates Java code for clients and services that wish to work in accordance with the contract stated in that WSDL. You can see these generated artifacts in **build/classes/com/my/fake** – since we didn’t tell it otherwise, **wsimport** came up with a base Java package derived from the WSDL’s **targetNamespace**. Feel free to review these files; we’ll start looking at them in more detail in the next three chapters.

An Abstract Model for Order Tracking

LAB 2A

12. If you're using Eclipse, you may want to review the WSDL editor's interpretation of what you've done. Click the **Design** tab to see your new port type:

Tracking		
⚙️ checkStatus		
➡️ input	ID	long
⬅️ output	Status	StatusEnum
⚙️ getDetail		
➡️ input	ID	long
⬅️ output	Order	Order
⚙️ cancel		
➡️ input	ID	long

13. Notice too that you can drill down on any of the XML Schema types used by the WSDL message parts, by clicking on the arrow symbols to the right of the port type depiction. This will open the corresponding schema document and take you to the definition of the appropriate element or type.
14. If you have extra time, you might even try re-building this port type, or a second one just like it, from scratch using the graphical editor. We won't provide step-by-step instructions for this, but start in the WSDL Design view and right-click to **Add Port Type**. You'll get the hang of it pretty quickly from there.

A Concrete Model for Order Tracking

LAB 2B

In this lab you will complete the WSDL document from Lab 2A by adding concrete information on service, port, and SOAP/HTTP binding. You will again validate your descriptor, and then you will use it in a full Ant build of the client application, using it to generate proxy classes on which a pre-written client application will depend for its SOAP messaging.

Lab workspace:	Labs/Lab02B
Backup of starter code:	Examples/Tracking/Step2
Answer folder(s):	Examples/Tracking/Step3
Files:	Client/WSDL/OrderTracking.wsdl Client/src/org/biz/Tracker.java

Instructions:

1. Build the Tracking service now from the **Service** directory.
2. Now, in the **Client** directory, open the starter file **OrderTracking.wsdl**. This is the completed version from the previous lab, and so includes the XML Schema and the abstract model: `<message>`, `<operation>`, and `<portType>` components. We're done with that lousy old **Fake.wsdl** now, and you'll build a proper concrete model into your own WSDL descriptor, and validate that instead.
3. So, 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 `<operation>`s 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 for this descriptor.) Give this element a **transport** attribute that identifies the HTTP transport by the URI shown below:

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

6. Set the attribute **style** to "rpc".

A Concrete Model for Order Tracking**LAB 2B**

7. As the second child of the binding element, add an `<operation>` with the name “checkStatus”. Give this both `<input>` and `<output>` child elements, and for each of these declare a single child element of the name `<soap:body>`. The body will define **use** as “literal”, and the **namespace** for the operation element – copy this URI from the **targetNamespace** for the WSDL descriptor itself, found at the top of the document.
8. Create an `<operation>` element for **getDetail**: other than the operation name, the content will be identical to the one you’ve just created for **checkStatus**.
9. Create a third `<operation>` element for **cancel**. This too is similar to **checkStatus**, except that as a one-way operation it will need no `<output>` binding.

This completes the binding descriptor; you now have described a concrete SOAP/HTTP messaging model.

10. Lastly you have to identify the service itself: create a `<service>` element with the name “Track”. Its one child element will be a `<port>` element with the name “Tracking” and a **binding** attribute identifying **order:TrackingBinding**.
11. As a child of this `<port>` element, create a `<soap:address>` element, whose **location** attribute has this value:

```
http://localhost:8080/Tracking/Track
```

Or, if you built the target service in Eclipse, use this instead:

```
http://localhost:8080/Labs_Lab02B_Service/Track
```

12. Validate your WSDL and fix any errors; as before, no news is good news ...

```
validate WSDL/OrderTracking.wsdl
```

13. Now we’ll try to get this li'l descriptor into action. Take a quick look at the pre-built client code in **Tracker.java**, and note that it connects to the service through a **Track** class. This proxy implements the web service interface so that it has one Java method for each operation.

```
Tracking service = new Track ().getTracking ();
```

If you go looking for **Track.class** and **Tracking.class**, you’ll find that they’ve been correctly generated to **build/classes/org/ebiz/ordertracking/wsdl** – again, based on the **targetNamespace**. **Tracker.java** will look for classes in package **org.biz.ws**, and the upcoming Ant build will make that happen, by invoking **wsimport** with the **-p** switch to dictate the Java package for generated classes.

14. Run **ant**; you might want to take a look now in **build/classes/org/biz/ws** to see the generated Java classes.

A Concrete Model for Order Tracking

LAB 2B

15. Try the client application and you should see results as follows:

```
run check 2
BACK_ORDERED
run detail 2
Order details:
  ID:          2
  Customer ID: 3456
  Status:      BACK_ORDERED
              100 Darjeeling
              25 Lapsang Souchong
run cancel 2
Order canceled.
run check 2
CANCELED
```

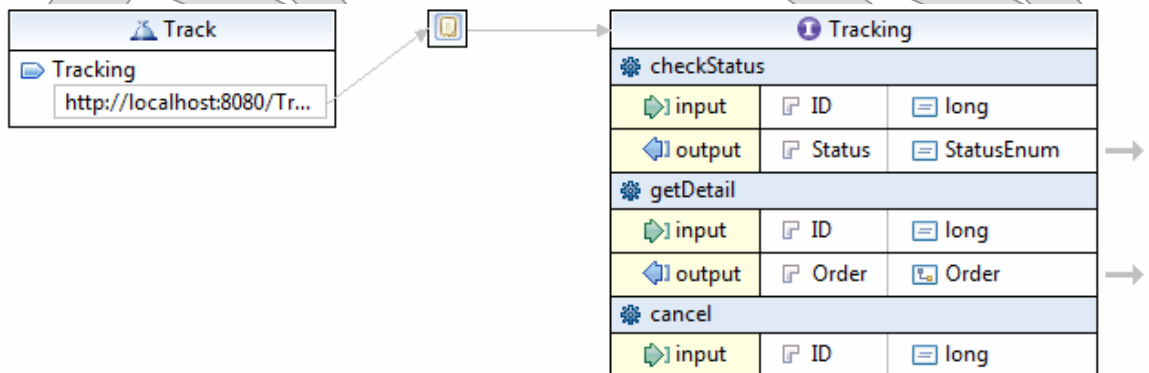
You can also try these commands through the SOAPSniffer – just add the extra command-line argument **-sniff**:

```
run -sniff check 2
BACK_ORDERED
```

This is one of a few addressing tricks we build into our JAX-WS client applications for the course; see Appendix A for details.

Optional Steps

16. See your finished WSDL in the Eclipse editor's Design view – there's not that much new here since the binding isn't really depicted, but it's worth a quick look:



17. Try running the **Dynamic** application as in the demo, pointing it at the WSDL:

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

or

```
http://localhost:8080/Labs_Lab02B_Service/Track?WSDL
```

A Concrete Model for Order Tracking

LAB 2B

Not all operations will work, and this shows the limitations of this little Java client application – it can only handle single-value parts, so it can't show a full order as returned by the **getDetail** operation, and it expects request/response operations so it's thrown off-balance by the one-way **cancel** operation.

