Problem: one of the most requested features for a Cisco Unified Contact Center Express (UCCX) script is to have an easy Web Services (WS) client (also known as SOAP client) implementation.

Some use various frameworks, like Apache Axis to achieve this; even though from the administration perspective this might not be a good solution, since both the framework libraries and the client code must be uploaded on to the UCCX platform. One should carefully watch the framework version and dependencies, making UCCX administration more complex.

Analysis: a rather extreme approach is not to use any custom libraries and classes, but to insert Java code directly into the script that crafts XML and communicates with the Web Services Server. This is not an elegant solution at all, since it requires detailed information about the internals (like the XML structure) of the web service itself, which was supposed to be hidden by the Web Services Object Oriented Programming (OOP) style.

Solution: the Java Development Kit (JDK) version 1.6 already contains the necessary libraries and tools to set up a WS client. In other words, all UCCX versions that are built on JDK 1.6 are perfectly capable of creating a lightweight OOP WS client, using the JAX-WS (aka "Metro") project libraries, which are built into the JDK itself.

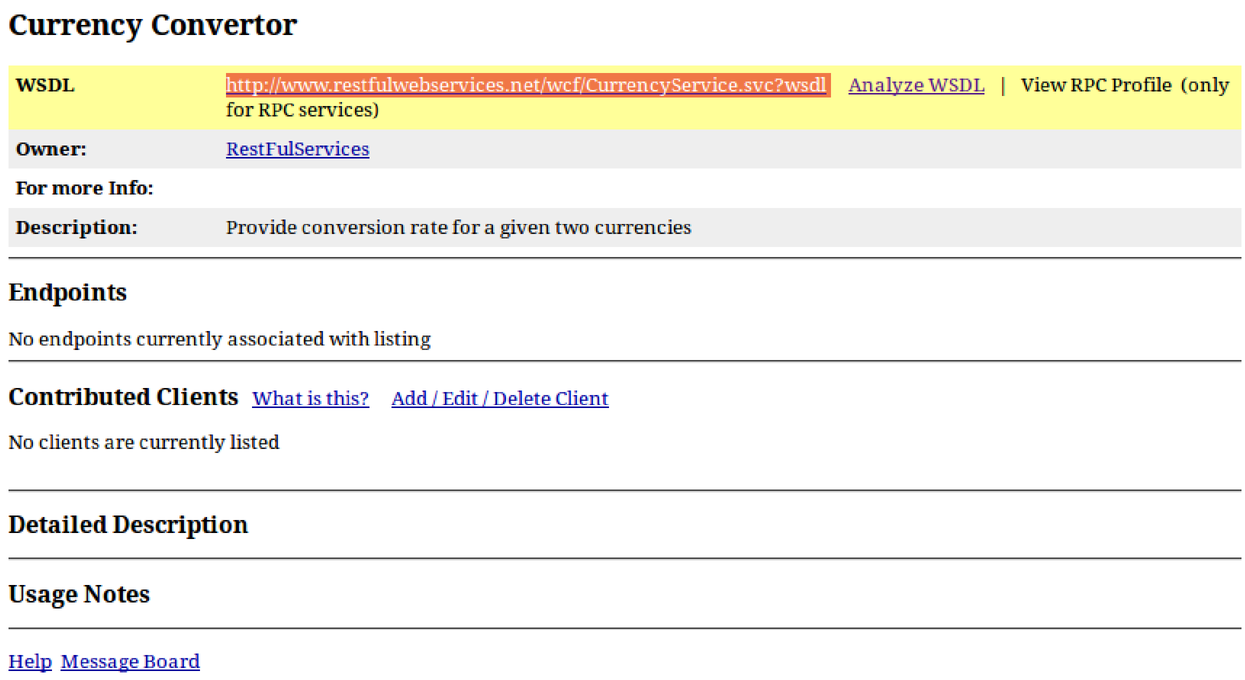
This step by step guide has been created to show an example of a WS client implementation in UCCX 8.5.

Please note:

- This is a "contract first" approach, assuming the WSDL is available (both while project compilation and runtime);
- JDK 1.6 is installed (avoid JDK 1.7, which is the latest JDK version in order to eliminate some compatiblity issues with UCCX);
- the Eclipse Integrated development environment (IDE) is used in this guide, however, any other Java-capable IDE should work.

This example uses a Web Service that is available at xmethods.net, namely RestFulServices's Currency Convertor.

1. First, get the WSDL URL from the web page mentioned above:



Currency Convertor

WSDL	http://www.restfulwebservices.net/wcf/CurrencyService.svc?wsdl Analyze WSDL | View RPC Profile (only for RPC services)
Owner:	RestFulServices
For more Info:	
Description:	Provide conversion rate for a given two currencies

Endpoints

No endpoints currently associated with listing

Contributed Clients

No clients are currently listed

Detailed Description

Usage Notes

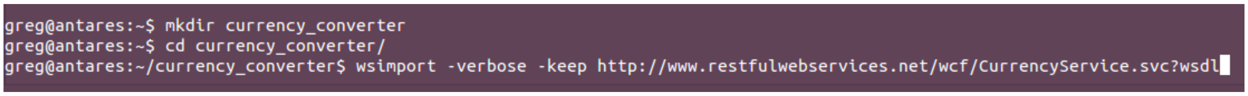
As we can see, no detailed Description or Usage Notes. No problem, we will figure it out.

2. Using the command line, generate the necessary files using the JDK wsimport tool.

Notice this guide has been written using a computer using Ubuntu Linux, however, the JDK on Windows or Mac does have the same libraries and tools, so don't worry.

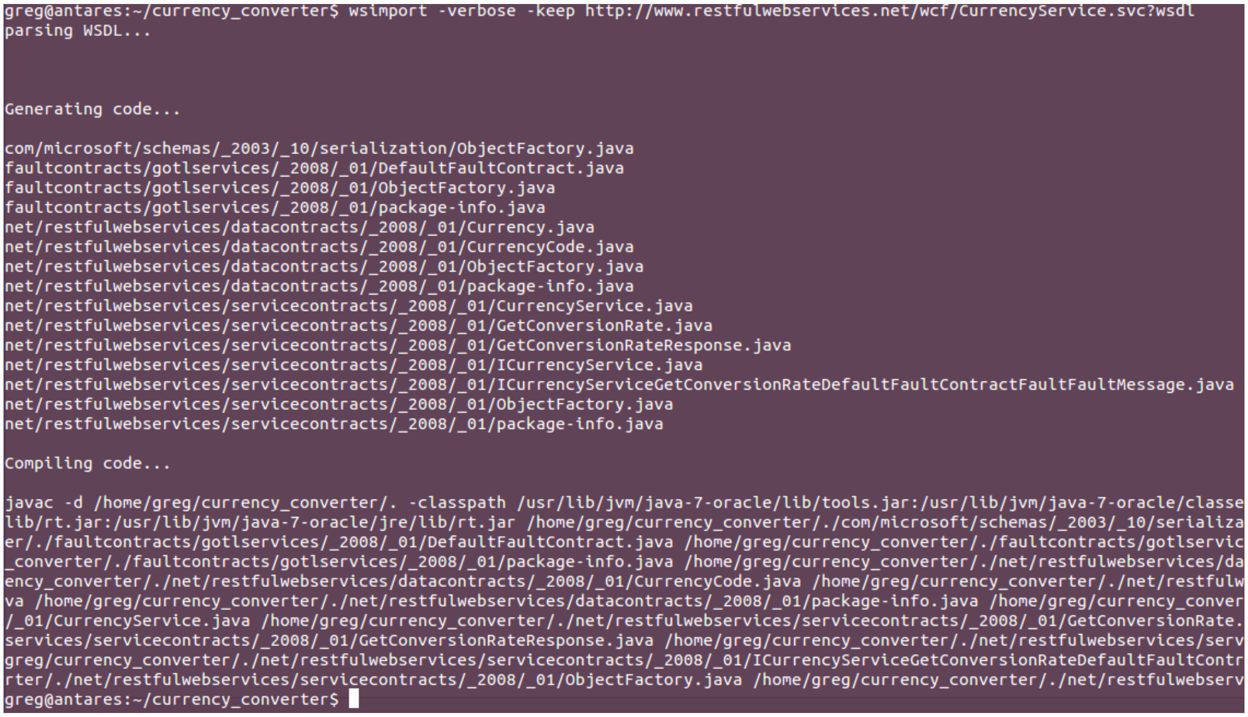
But first, let's just create a new directory for the files: in this example, `currency_converter`.

Then issue the `wsimport` command with the following switches: `-verbose` (to see detailed messages), `-keep` (to keep Java source files, just in case we wanted to dig deeper) and finally, the WSDL URL itself.



```
greg@antares:~$ mkdir currency_converter
greg@antares:~$ cd currency_converter/
greg@antares:~/currency_converter$ wsimport -verbose -keep http://www.restfulwebservices.net/wcf/CurrencyService.svc?wsdl
```

After hitting Enter, wsimport prints out status messages.



```
greg@antares:~/currency_converter$ wsimport -verbose -keep http://www.restfulwebservices.net/wcf/CurrencyService.svc?wsdl
parsing WSDL...



Generating code...

com/microsoft/schemas/_2003/_10/serialization/ObjectFactory.java
faultcontracts/gotlservices/_2008/_01/DefaultFaultContract.java
faultcontracts/gotlservices/_2008/_01/ObjectFactory.java
faultcontracts/gotlservices/_2008/_01/package-info.java
net/restfulwebservices/datacontracts/_2008/_01/Currency.java
net/restfulwebservices/datacontracts/_2008/_01/CurrencyCode.java
net/restfulwebservices/datacontracts/_2008/_01/ObjectFactory.java
net/restfulwebservices/datacontracts/_2008/_01/package-info.java
net/restfulwebservices/servicecontracts/_2008/_01/CurrencyService.java
net/restfulwebservices/servicecontracts/_2008/_01/GetConversionRate.java
net/restfulwebservices/servicecontracts/_2008/_01/GetConversionRateResponse.java
net/restfulwebservices/servicecontracts/_2008/_01/ICurrencyService.java
net/restfulwebservices/servicecontracts/_2008/_01/ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage.java
net/restfulwebservices/servicecontracts/_2008/_01/ObjectFactory.java
net/restfulwebservices/servicecontracts/_2008/_01/package-info.java

Compiling code...

javac -d /home/greg/currency_converter/. -classpath /usr/lib/jvm/java-7-oracle/lib/tools.jar:/usr/lib/jvm/java-7-oracle/classe
lib/rt.jar:/usr/lib/jvm/java-7-oracle/jre/lib/rt.jar /home/greg/currency_converter/./com/microsoft/schemas/_2003/_10/serializa
er/./faultcontracts/gotlservices/_2008/_01/DefaultFaultContract.java /home/greg/currency_converter/./faultcontracts/gotlservic
_converter/./faultcontracts/gotlservices/_2008/_01/package-info.java /home/greg/currency_converter/./net/restfulwebservices/da
ency_converter/./net/restfulwebservices/datacontracts/_2008/_01/CurrencyCode.java /home/greg/currency_converter/./net/restfulw
va /home/greg/currency_converter/./net/restfulwebservices/datacontracts/_2008/_01/package-info.java /home/greg/currency_conver
/_01/CurrencyService.java /home/greg/currency_converter/./net/restfulwebservices/servicecontracts/_2008/_01/GetConversionRate.
services/servicecontracts/_2008/_01/GetConversionRateResponse.java /home/greg/currency_converter/./net/restfulwebservices/serv
greg/currency_converter/./net/restfulwebservices/servicecontracts/_2008/_01/ICurrencyServiceGetConversionRateDefaultFaultContr
rter/./net/restfulwebservices/servicecontracts/_2008/_01/ObjectFactory.java /home/greg/currency_converter/./net/restfulwebserv
greg@antares:~/currency_converter$ █
```

If it ended with "Compiling code..." and without any error messages, then all the necessary artefacts have been created.

Now let's just stop for a moment to see, what we have done so far.

Actually, these "artefacts" are simple Java class files, representing the objects necessary to build a WS client that conforms to the specification in the WSDL which serves as a contract.

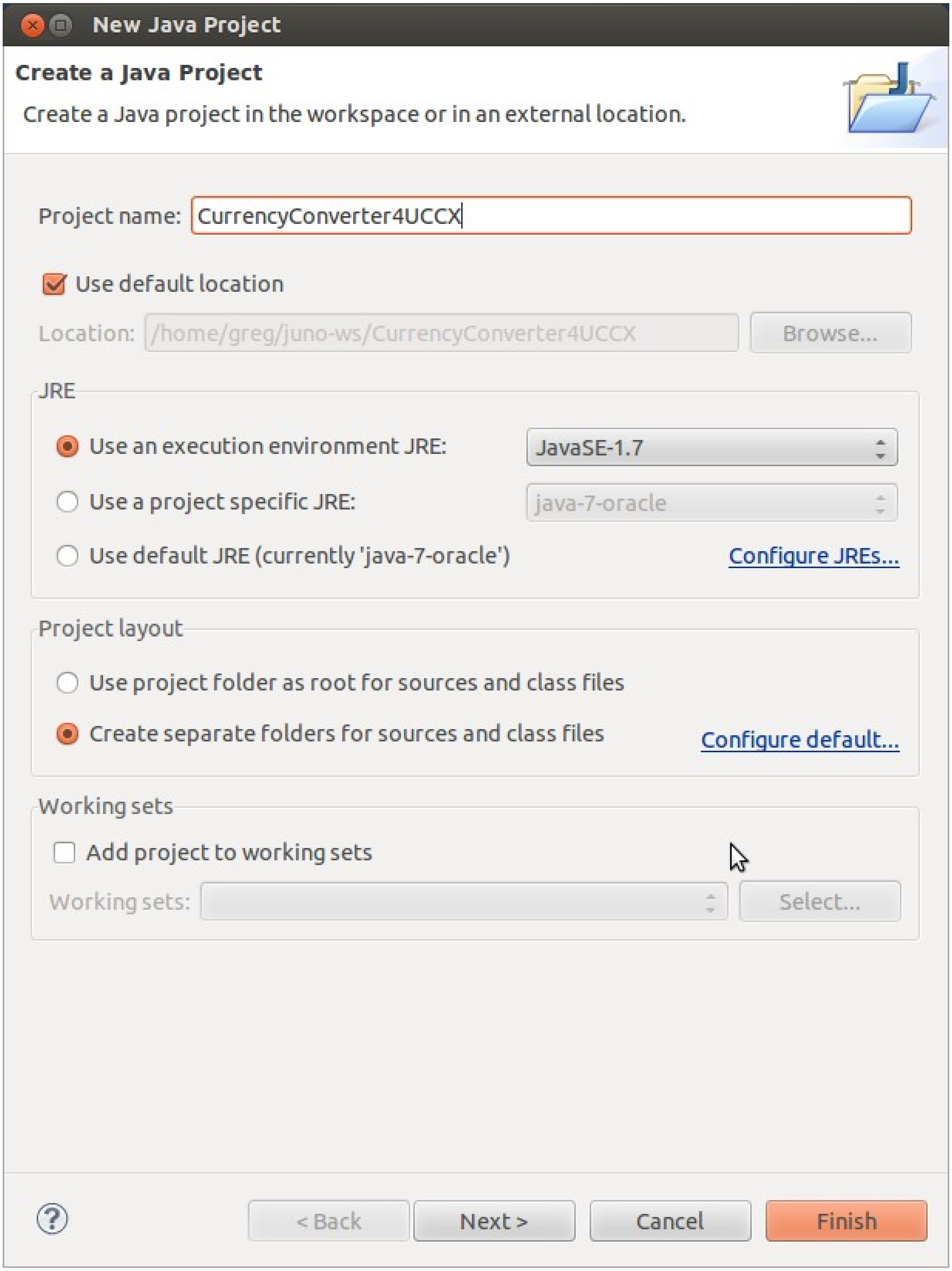
Please note, we are about to write a program in Java, even though the web server is built in using a .NET language. This is one of the strengths of the Web Services approach: object oriented programming, using a transparent communication method (XML, HTTP) where the client and the server may be built on a completely different platform.

Actually, this is all. The only thing you have to do is to upload these classes to the UCCX, do the necessary reloads and you can use these artefacts right away in your scripts.

In order to achieve better understanding, it's strongly recommended to test the WS functionality in a simple Java program.

Why we need an IDE: for its content assist technology. Remember, at this point we don't have any clues what methods are available and what methods should be use.

3. Start up Eclipse and create a new Java project, for instance, `CurrencyConverter4UCCX`:



Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: CurrencyConverter4UCCX

☑ Use default location

Location: /home/greg/juno-ws/CurrencyConverter4UCCX

JRE

● Use an execution environment JRE: JavaSE-1.7

○ Use a project specific JRE: java-7-oracle

○ Use default JRE (currently 'java-7-oracle')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

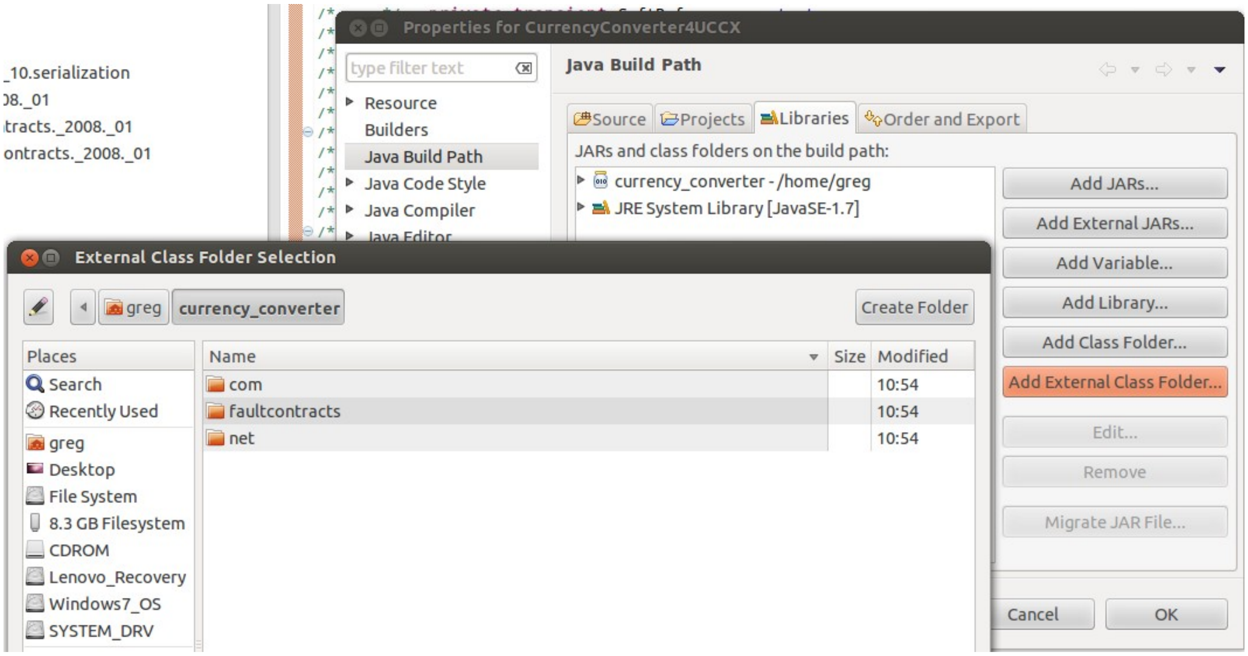
Working sets

☐ Add project to working sets

Working sets:

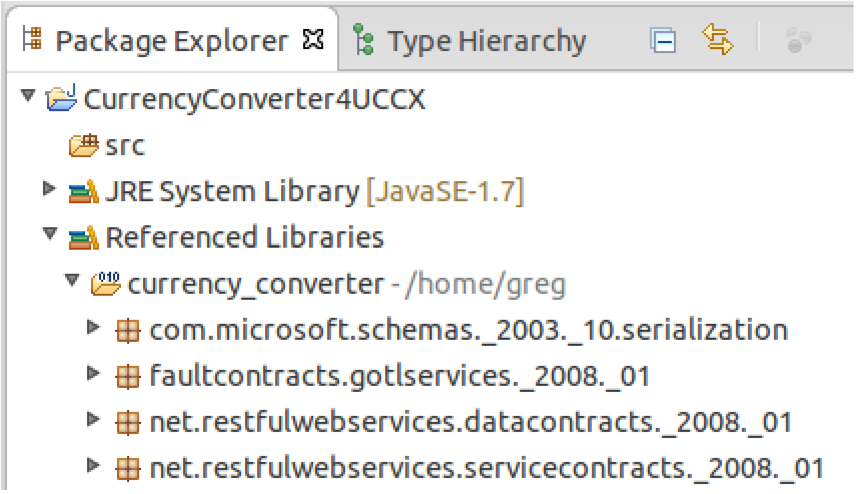
Then add the artefacts we created with the wsimport tool to the project so our little Java program can use them. Right click the project name, choose Properties; then Java Build Path, Libraries tab. Click the

Add External Class Folder... button.



We just add the `currency_converter` directory we created a while ago; Eclipse is intelligent enough to include all the subdirectories and files there.

In the Package Explorer tab, if you expand the Referenced Libraries, you should see this hierarchy:



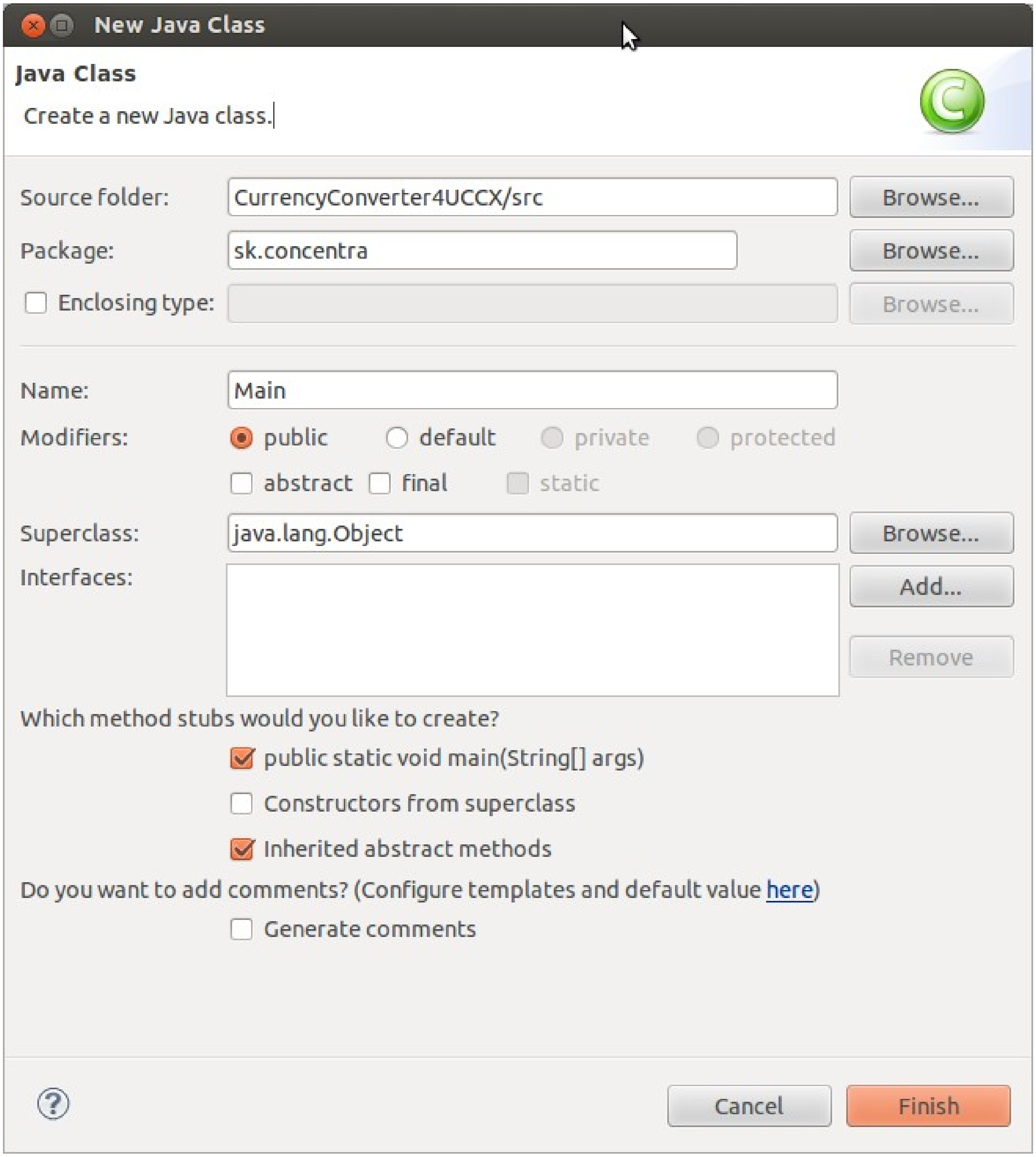
CurrencyConverter4UCCX
  src
  JRE System Library [JavaSE-1.7]
  Referenced Libraries
    currency_converter - /home/greg
      com.microsoft.schemas._2003._10.serialization
      faultcontracts.gotlservices._2008._01
      net.restfulwebservices.datacontracts._2008._01
      net.restfulwebservices.servicecontracts._2008._01

Don't worry about their contents yet, we will get to that later.

4. Create a new Java class, in a new package. In this example, I create a new Java class called `Main`, in the `sk.concentra` package. This means the qualified name of the class is `sk.concentra.Main`. Think of the term "qualified name" as the "fully qualified domain name" (FQDN), or the full name of a person. You might have noticed if you insert custom Java code in a UCCX script, you always use

qualified names for types.

How to create a new class: In Package Explorer, right click `CurrencyConverter4UCCX`. New > Class. Fill in the Package (for instance: sk.concentra) and Name Fields (for instance: Main). Check the Which method stubs would you like to create? "public static void main(String[] args)", like this:



New Java Class

Java Class

Create a new Java class.

Source folder: CurrencyConverter4UCCX/src

Package: sk.concentra

☐ Enclosing type:

Name: Main

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☑ public static void main(String[] args)

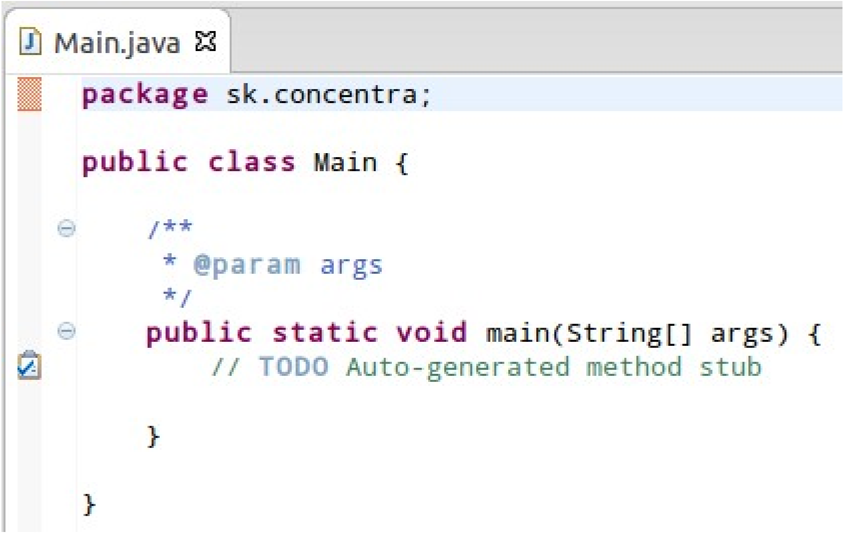
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

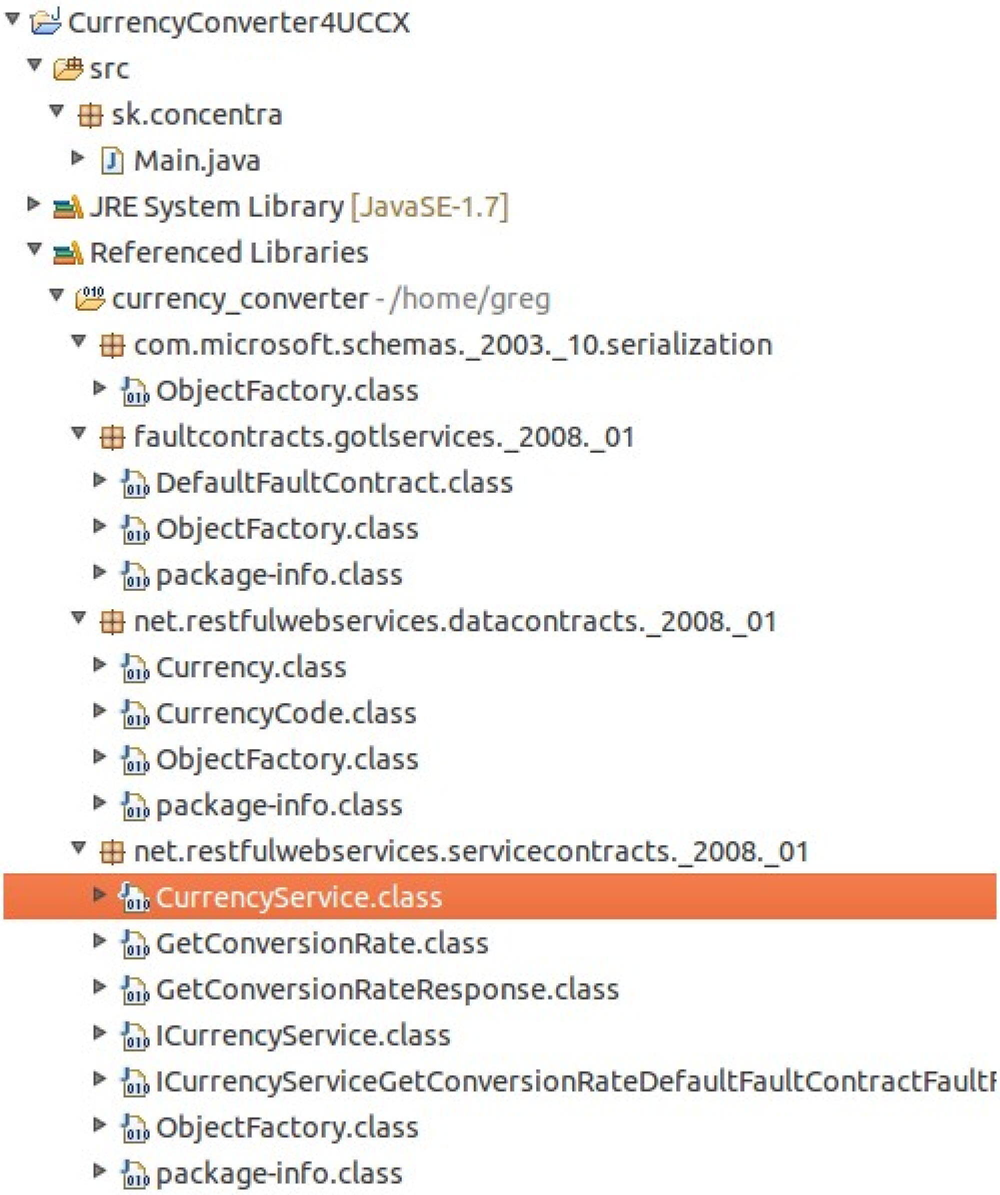
☐ Generate comments

And, as we can see, a new Java class skeleton has been created for us:



```
package sk.concentra;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

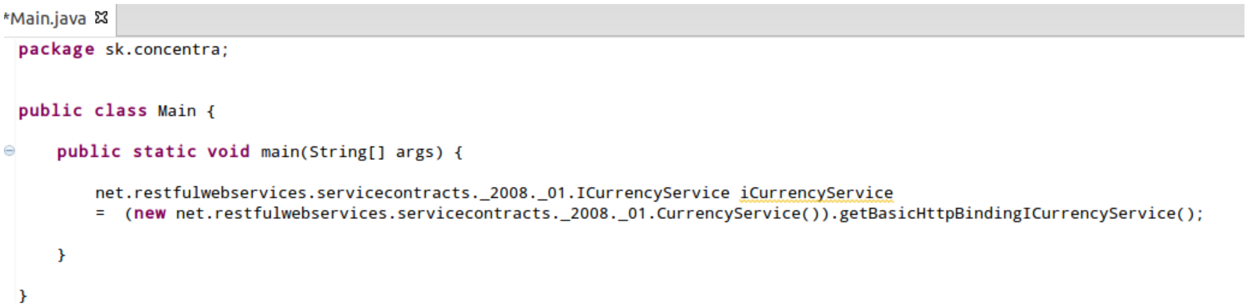
5. Now expand all the packages in Referenced Libraries (in the Package Explorer tab) and look for a pair of classes that are named `I..Service` and `...Service` where `...` is... well, something. In this case, we can clearly see one pair: `CurrencyService` and `ICurrencyService`:



- CurrencyConverter4UCCX
  - src
    - sk.concentra
      - Main.java
  - JRE System Library [JavaSE-1.7]
  - Referenced Libraries
    - currency_converter - /home/greg
      - com.microsoft.schemas._2003._10.serialization
        - ObjectFactory.class
      - faultcontracts.gotlservices._2008._01
        - DefaultFaultContract.class
        - ObjectFactory.class
        - package-info.class
      - net.restfulwebservices.datacontracts._2008._01
        - Currency.class
        - CurrencyCode.class
        - ObjectFactory.class
        - package-info.class
      - net.restfulwebservices.servicecontracts._2008._01
        - CurrencyService.class
        - GetConversionRate.class
        - GetConversionRateResponse.class
        - ICurrencyService.class
        - ICurrencyServiceGetConversionRateDefaultFaultContractFault
        - ObjectFactory.class
        - package-info.class

Now, take a deep breath and write their qualified names into Main.java's main method. Actually, let's start right away with this:

```
net.restfulwebservices.servicecontracts._2008._01.ICurrencyService
iCurrencyService
    =  (new
net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).
getBasicHttpBindingICurrencyService();
```



```
package sk.concentra;

public class Main {

    public static void main(String[] args) {

        net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
        =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();

    }

}
```

You just created a new variable, named `iCurrencyService`, of type `net.restfulwebservices.servicecontracts._2008._01.ICurrencyService`.

Its value is based on a new instance of the class `net.restfulwebservices.servicecontracts._2008._01.CurrencyService`; and you called `getBasicHttpBindingICurrencyService()` method on this new instance, which actually returns a variable, of type `ICurrencyService`.

Now, I owe you some explanation again.

`CurrencyService` is the *implementation* ("It works like this").

`ICurrencyService` is an *interface* ("It should work like this").

`CurrencyService` actually implements `ICurrencyService`, with all the nuts and bolts. `ICurrencyService` is just a contract that contains how the web service should be implemented.

So, when we create a new instance of the implementation, we already have a working implementation of that web service. But it's not what we want, a full implementation that we can't control. So we use its method (in this case, `getBasicHttpBindingICurrencyService()`), to get the contract, so we can implement our own WS client.

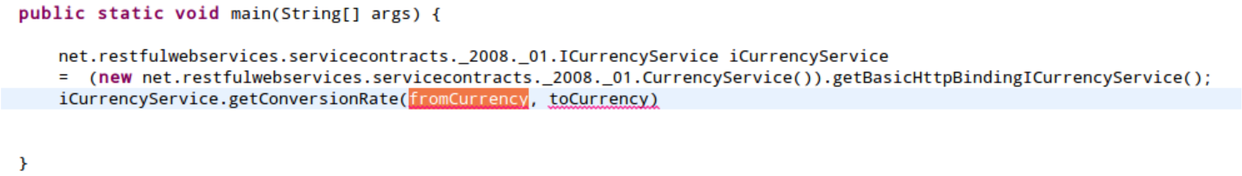
It's kind of against the logic, I know, but it's the way we go. A rule of a thumb: use the content assist feature of Eclipse (or your favorite IDE) to see, what methods are available on the implementation that return the type of the interface. Chances are that the method name would contain the expressions "get", "http" or even "port".

6. So we have the `iCurrencyService`, which is again, the "contract". It contains the methods available on the WS server.

At this moment, we will use the Content Assist feature of Eclipse. Write down the word `iCurrencyService`, then dot (.) and wait for the Content Assist window to appear. If it doesn't, make it appear (in Eclipse, Ctrl-Space).

As we can see, there's only one method, `getConversionRate` (your WS, of course, may contain more methods). By pressing Enter, Content Assist will insert this method, with the required variables.

At this point, our Java program looks like this:



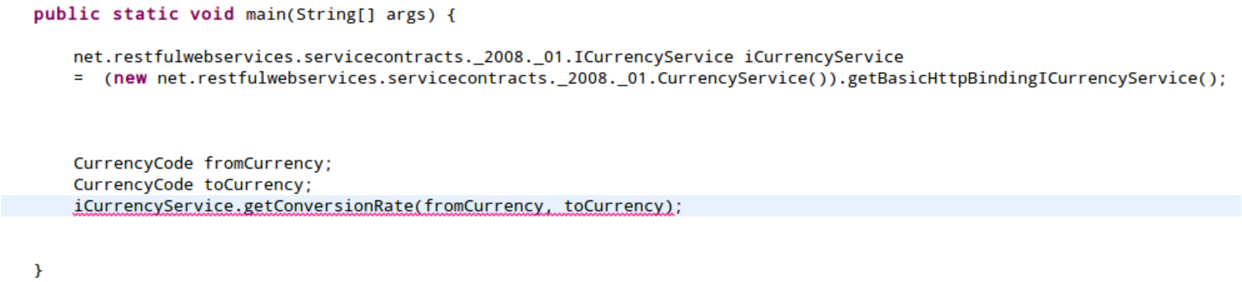
```
public static void main(String[] args) {

    net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
    =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
    iCurrencyService.getConversionRate(fromCurrency, toCurrency)


}
```

That's right, it's underlined red because of an error. It wants those parameters badly. And of course, they should exist before we actually call this method. Make some room *before* this row. Now just double click that `fromCurrency` so it gets selected and hit Ctrl-1 (Quick fix). Let's see what quick fix is available.

Create a local variable: that's what we are looking for.

Let's do this both for `fromCurrency` and `toCurrency`.

You should end up with something like this:



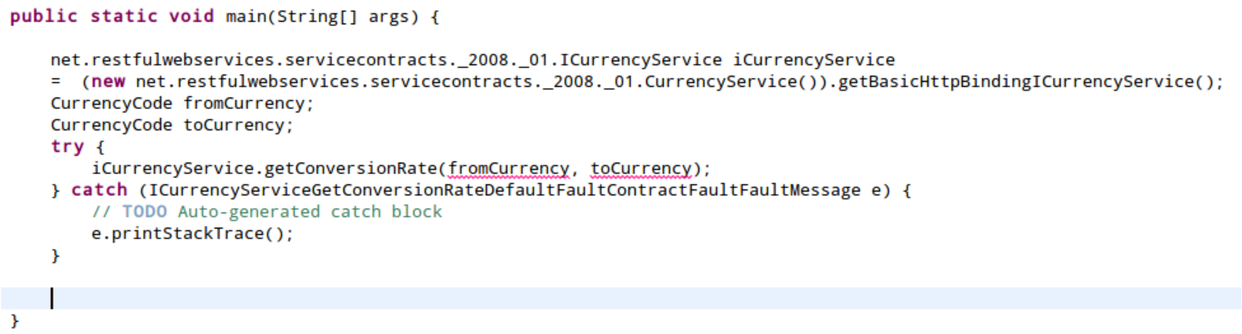
```
public static void main(String[] args) {

    net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
    =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();



    CurrencyCode fromCurrency;
    CurrencyCode toCurrency;
    iCurrencyService.getConversionRate(fromCurrency, toCurrency);


}
```

Also, don't forget the fact that all Java commands should end with a semicolon (;).

Hmm, it's underlined again. What is it this time? Let's just double click the whole thing and again, press Ctrl-1 for quick Fix.

This time, let's use that little lightbulb with a red and white cross to see what's wrong and what should be fixed:

It'll tell you an exception should be caught. So let's do that: surround with try and catch. Now it looks like this:



```
public static void main(String[] args) {

    net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
    =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
    CurrencyCode fromCurrency;
    CurrencyCode toCurrency;
    try {
        iCurrencyService.getConversionRate(fromCurrency, toCurrency);
    } catch (ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

And it still complains about what? `fromCurrency` and `toCurrency` are not initialized.

Let's just do this for now: assign `null` to them.

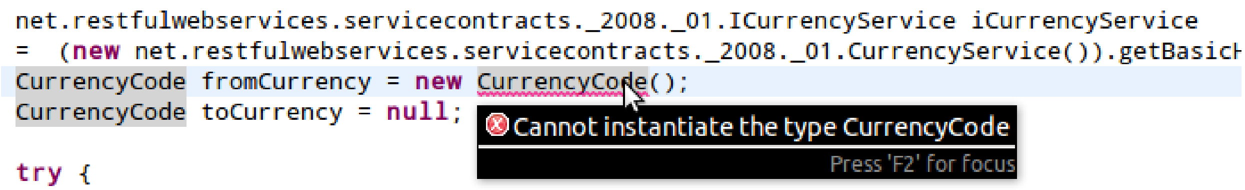
```
CurrencyCode fromCurrency = null;
CurrencyCode toCurrency = null;
```

This clears the error, but does not satisfy our needs. We don't want to send nulls. We want to send something useful.

Let's explore a few things.

Both `fromCurrency` and `toCurrency` are of type `CurrencyCode`. Can this `CurrencyCode` be instantiated?

Let's see:

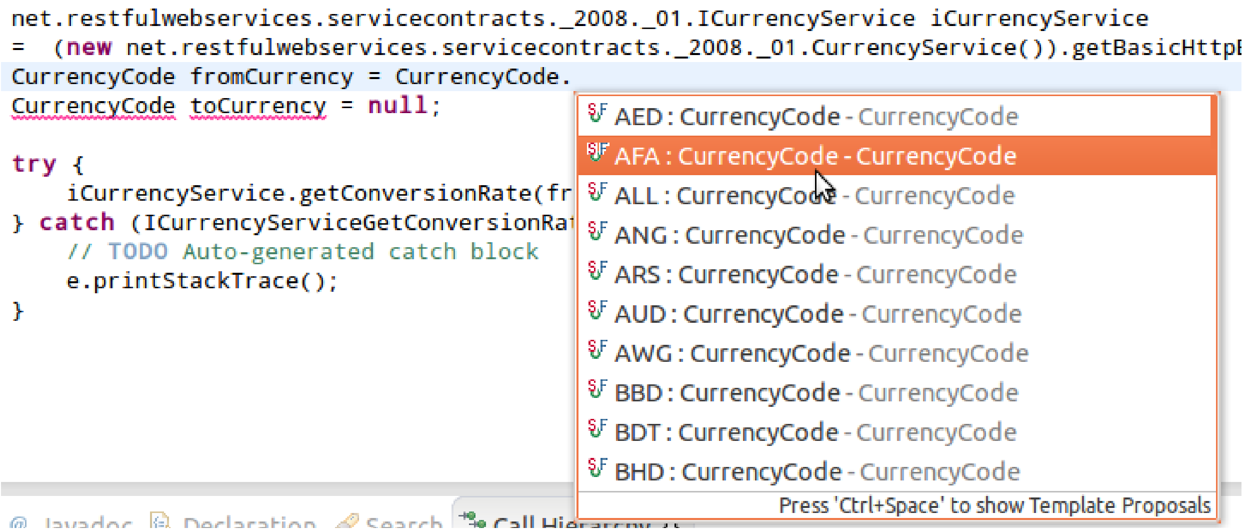


```
net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
=  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicH
CurrencyCode fromCurrency = new CurrencyCode();
CurrencyCode toCurrency = null;

try {
```

Cannot instantiate the type CurrencyCode

No.

Well, are there any fields we can use? Let's see:



```
net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
=  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttp
CurrencyCode fromCurrency = CurrencyCode.
CurrencyCode toCurrency = null;

try {
    iCurrencyService.getConversionRate(fr
} catch (ICurrencyServiceGetConversionRat
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

AED : CurrencyCode - CurrencyCode
AFA : CurrencyCode - CurrencyCode
ALL : CurrencyCode - CurrencyCode
ANG : CurrencyCode - CurrencyCode
ARS : CurrencyCode - CurrencyCode
AUD : CurrencyCode - CurrencyCode
AWG : CurrencyCode - CurrencyCode
BBD : CurrencyCode - CurrencyCode
BDT : CurrencyCode - CurrencyCode
BHD : CurrencyCode - CurrencyCode
Press 'Ctrl+Space' to show Template Proposals

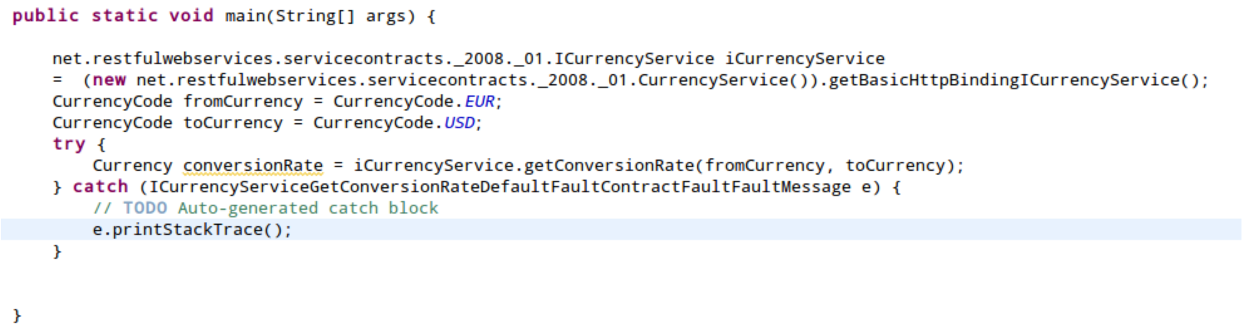
Yes, it does. (Actually, `CurrencyCode` is an *enum*):

Let's do the same thing both for `fromCurrency` and `toCurrency`. For instance, the `fromCurrency` be EUR (Euros) and the `toCurrency` USD (US Dollars):

```
CurrencyCode fromCurrency = CurrencyCode.EUR;
CurrencyCode toCurrency = CurrencyCode.USD;
```

One more thing: the `getConversionRate` method is called, but not returning anything. Let's fix that.

Let's just doubleclick and select `getConversionRate` method, hit Ctrl-1 for Quick Fix, and use "Assign to new local variable").

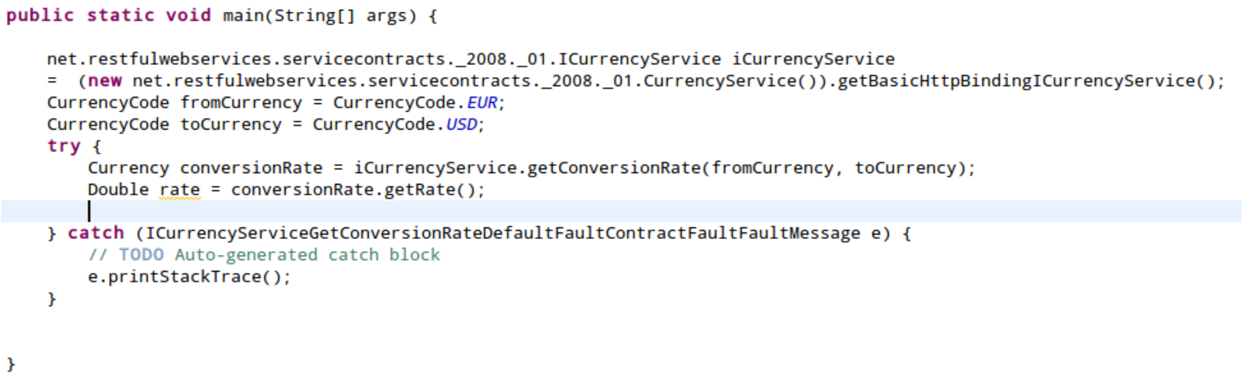


```
public static void main(String[] args) {

    net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
    =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
    CurrencyCode fromCurrency = CurrencyCode.EUR;
    CurrencyCode toCurrency = CurrencyCode.USD;
    try {
        Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
    } catch (ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }


}
```

Alright, now just make it write out that `conversionRate`, right?

Make some room after that `Currency conversionRate etc.` row, and write down the variable `conversionRate`. Again, wait for the content assist. If it doesn't appear, make it appear using Ctrl-Space.

That `getRate()` method looks promising. It even tells you its type: Double. Let's just make it complete by assigning the method result to a Double type variable then:

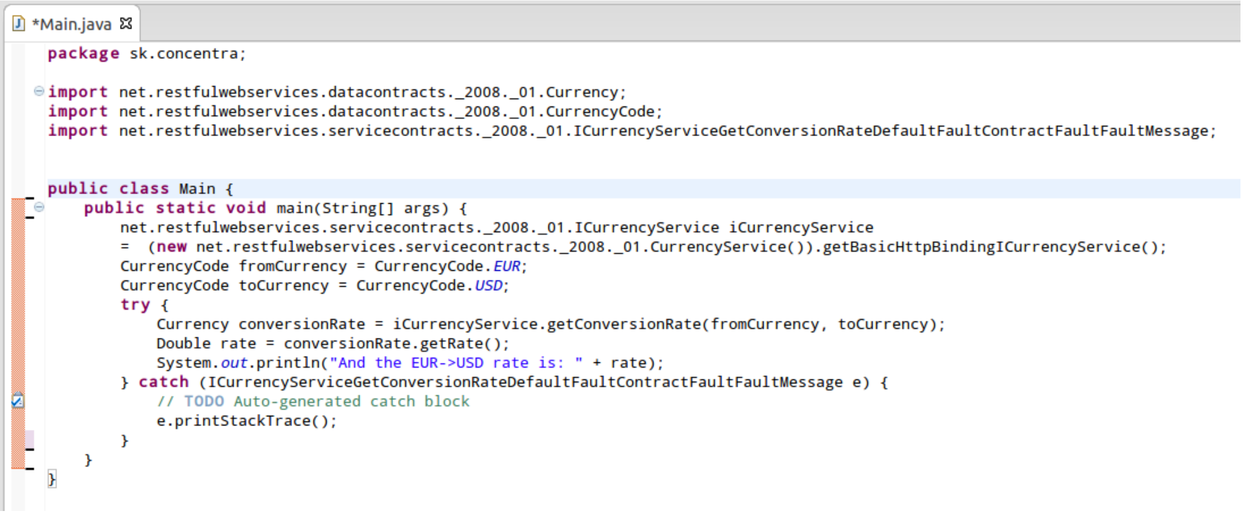


```
public static void main(String[] args) {

    net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
    =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
    CurrencyCode fromCurrency = CurrencyCode.EUR;
    CurrencyCode toCurrency = CurrencyCode.USD;
    try {
        Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
        Double rate = conversionRate.getRate();

    } catch (ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }


}
```

Let's just use a simple System.out.println to see what we got:

```
System.out.println("And the EUR->USD rate is: " + rate);
```

Let's clean things up a bit, so we have a nicely formatted code.



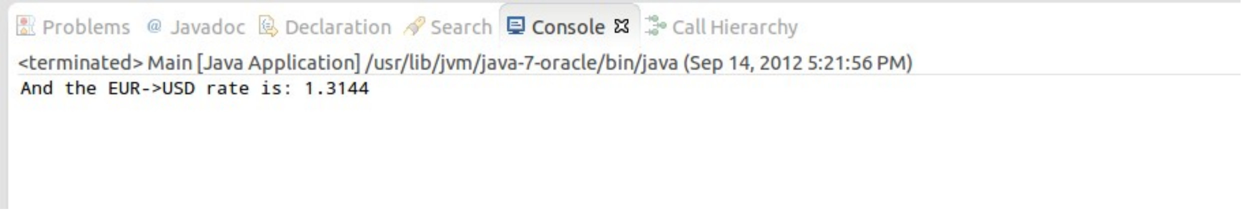
```
package sk.concentra;

import net.restfulwebservices.datacontracts._2008._01.Currency;
import net.restfulwebservices.datacontracts._2008._01.CurrencyCode;
import net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage;


public class Main {
    public static void main(String[] args) {
        net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
        =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
        CurrencyCode fromCurrency = CurrencyCode.EUR;
        CurrencyCode toCurrency = CurrencyCode.USD;
        try {
            Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
            Double rate = conversionRate.getRate();
            System.out.println("And the EUR->USD rate is: " + rate);
        } catch (ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Of course, don't forget to save it.

Now, time for some serious work for our little program.

Run it: click that Main.java file and then click that Play button.

Now wait... and watch the console tab.



```
<terminated> Main [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (Sep 14, 2012 5:21:56 PM)
And the EUR->USD rate is: 1.3144
```

If you can see this, grab a beer or a bottle of wine or a glass of milk or <your preferred drink> to celebrate with.

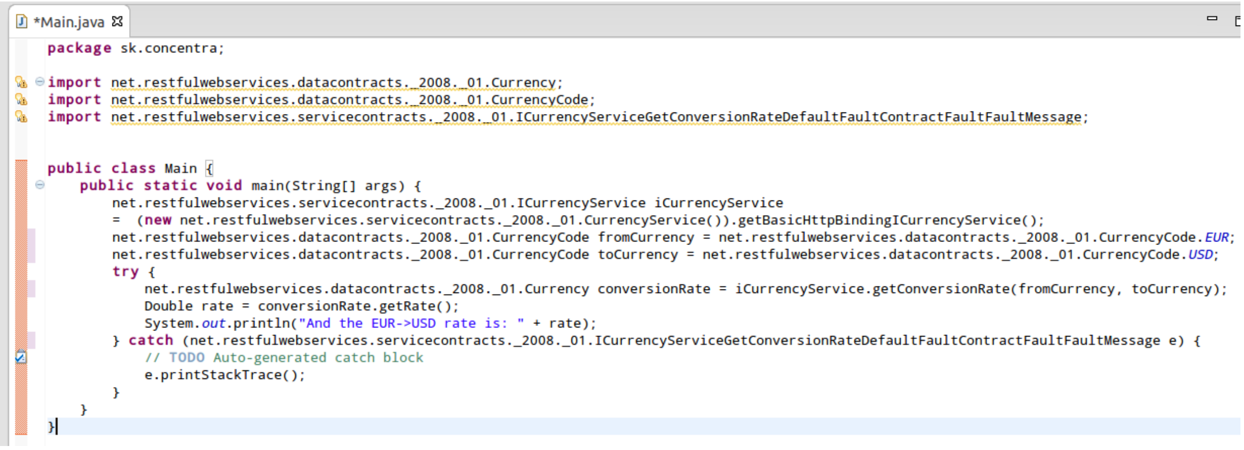
There are still two things remaining if we want to have the same functionality in UCCX:

- package the classes we need into one jar,
- types should be fully qualified so we can use them in UCCX.

Let's start with the more difficult stuff:

Select each type in the main method, that has been imported. Click "Copy Qualified name". And then just press Ctrl-V to replace it.

You should end up with something like this.

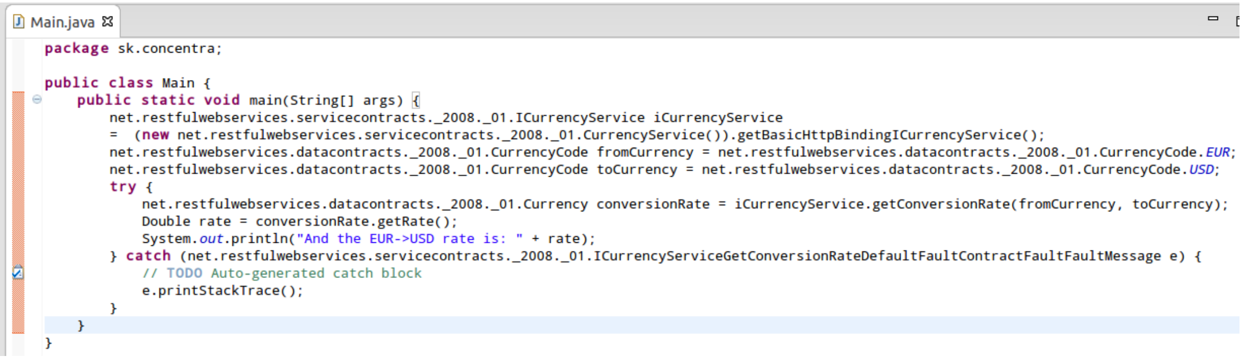


```
package sk.concentra;

import net.restfulwebservices.datacontracts._2008._01.Currency;
import net.restfulwebservices.datacontracts._2008._01.CurrencyCode;
import net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage;


public class Main {
    public static void main(String[] args) {
        net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
        =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
        net.restfulwebservices.datacontracts._2008._01.CurrencyCode fromCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.EUR;
        net.restfulwebservices.datacontracts._2008._01.CurrencyCode toCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.USD;
        try {
            net.restfulwebservices.datacontracts._2008._01.Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
            Double rate = conversionRate.getRate();
            System.out.println("And the EUR->USD rate is:  " + rate);
        } catch (net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Imports are unused, so let's just delete them. Finally, this is what you should see:

```
package sk.concentra;


public class Main {

  public static void main(String[] args) {

net.restfulwebservices.servicecontracts._2008._01.ICurrencyService
iCurrencyService

    =  (new
net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).
getBasicHttpBindingICurrencyService();

    net.restfulwebservices.datacontracts._2008._01.CurrencyCode
fromCurrency =
net.restfulwebservices.datacontracts._2008._01.CurrencyCode.EUR;

    net.restfulwebservices.datacontracts._2008._01.CurrencyCode
toCurrency =
net.restfulwebservices.datacontracts._2008._01.CurrencyCode.USD;

    try {

      net.restfulwebservices.datacontracts._2008._01.Currency
conversionRate = iCurrencyService.getConversionRate(fromCurrency,
toCurrency);

      Double rate = conversionRate.getRate();

      System.out.println("And the EUR->USD rate is:  " + rate);
    } catch
(net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGe
tConversionRateDefaultFaultContractFaultFaultMessage  e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```

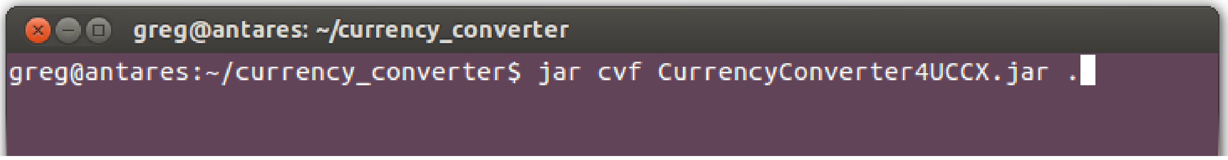
Or, if you prefer a screenshot:



```
package sk.concentra;

public class Main {
    public static void main(String[] args) {
        net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
        =  (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
        net.restfulwebservices.datacontracts._2008._01.CurrencyCode fromCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.EUR;
        net.restfulwebservices.datacontracts._2008._01.CurrencyCode toCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.USD;
        try {
            net.restfulwebservices.datacontracts._2008._01.Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
            Double rate = conversionRate.getRate();
            System.out.println("And the EUR->USD rate is: " + rate);
        } catch (net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

You will actually need this code in your UCCX script.

7. Package the artefacts into one JAR file. I prefer to do this using the command line:



```
greg@antares:~/currency_converter$ jar cvf CurrencyConverter4UCCX.jar .
```

This means, in the directory we created earlier, that holds the necessary artefacts, we create a JAR file named CurrencyConverter4UCCX.jar:

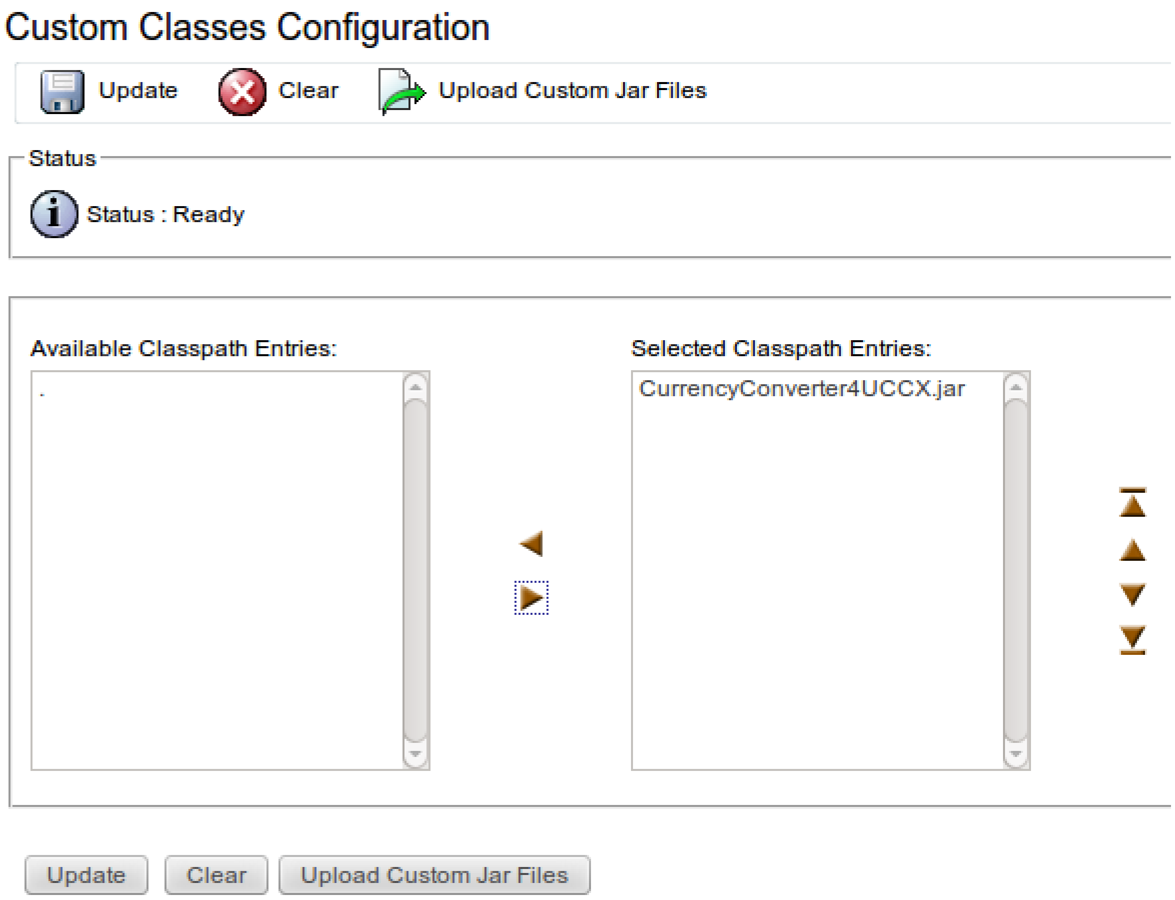
```
jar cvf CurrencyConverter4UCCX.jar .
```

(That last dot represents the working directory.)

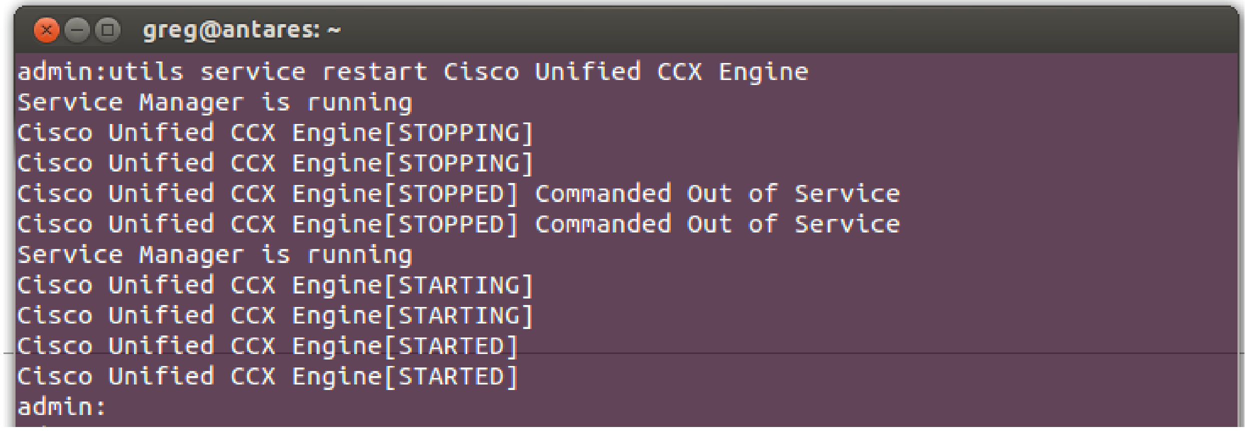
Voila, we've got our JAR file, named CurrencyConverter4UCCX.jar holding all the required helper classes to run the same program we tried a while ago in Eclipse.

8. So let's upload this JAR in UCCX. Using the Application Administration web, go to System > Custom File Configuration and then click Upload Custom Jar Files, then Upload Documents (Or, upload Documents, default language, classpath folder). Upload the CurrencyConverter4UCCX.jar file we created.

Then go back to the Custom Classes configuration page and make sure the JAR file is "Selected Classpath Entry":

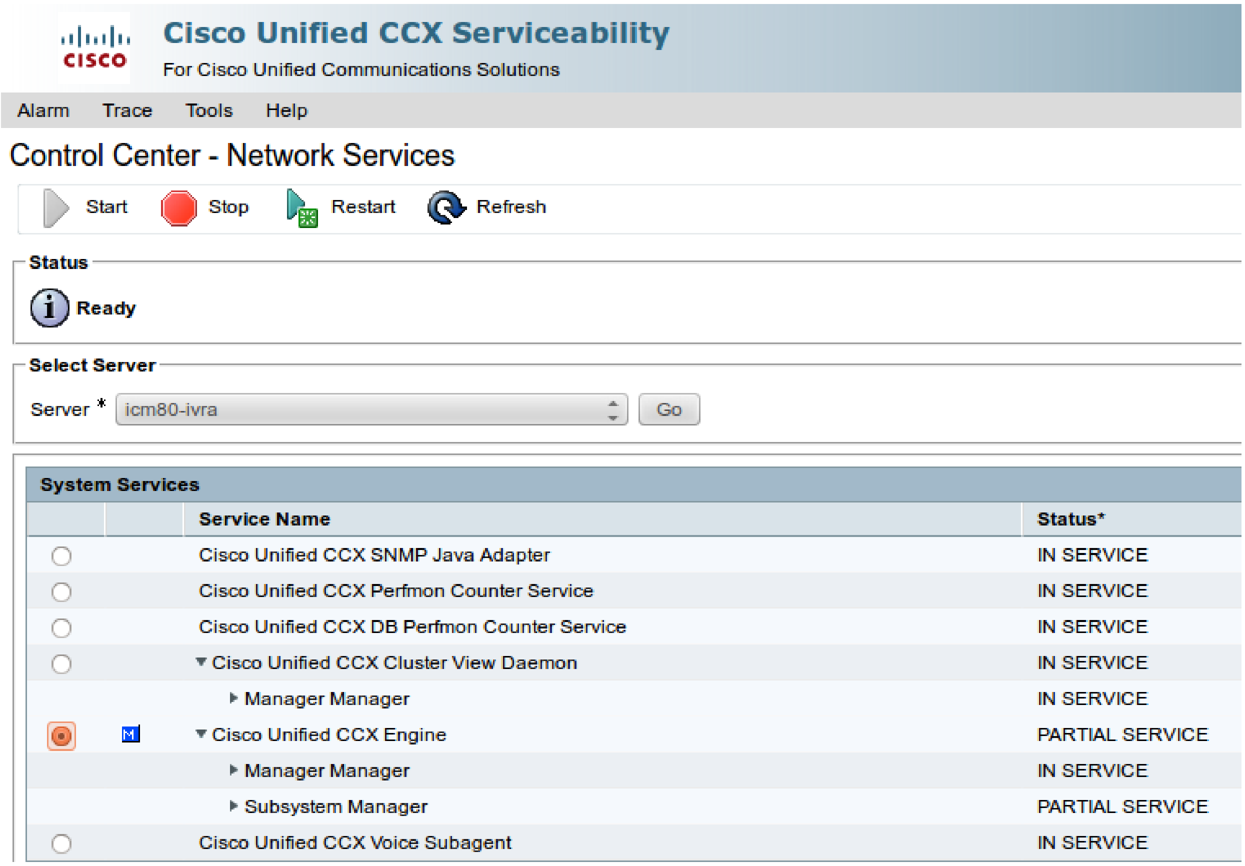

Custom Classes Configuration
Update
Clear
Upload Custom Jar Files
Status
Status : Ready
Available Classpath Entries:
.
Selected Classpath Entries:
CurrencyConverter4UCCX.jar
Update
Clear
Upload Custom Jar Files

You must restart the CCX Engine. The quickest way is to ssh into the UCCX and issue the `utils service <servicename>` command, like this:



```
admin:utils service restart Cisco Unified CCX Engine
Service Manager is running
Cisco Unified CCX Engine[STOPPING]
Cisco Unified CCX Engine[STOPPING]
Cisco Unified CCX Engine[STOPPED] Commanded Out of Service
Cisco Unified CCX Engine[STOPPED] Commanded Out of Service
Service Manager is running
Cisco Unified CCX Engine[STARTING]
Cisco Unified CCX Engine[STARTING]
Cisco Unified CCX Engine[STARTED]
Cisco Unified CCX Engine[STARTED]
admin:
```

Alternatively, log into the Unified CCX Serviceability page and restart the engine from there:



Cisco Unified CCX Serviceability
For Cisco Unified Communications Solutions
Alarm
Trace
Tools
Help
Start
Stop
Restart
Refresh
Go

Control Center - Network Services

Status

Ready

Select Server

Server * icm80-ivra

System Services

		Service Name	Status*
○		Cisco Unified CCX SNMP Java Adapter	IN SERVICE
○		Cisco Unified CCX Perfmon Counter Service	IN SERVICE
○		Cisco Unified CCX DB Perfmon Counter Service	IN SERVICE
○		▾ Cisco Unified CCX Cluster View Daemon	IN SERVICE
		▸ Manager Manager	IN SERVICE
●	M	▾ Cisco Unified CCX Engine	PARTIAL SERVICE
		▸ Manager Manager	IN SERVICE
		▸ Subsystem Manager	PARTIAL SERVICE
○		Cisco Unified CCX Voice Subagent	IN SERVICE

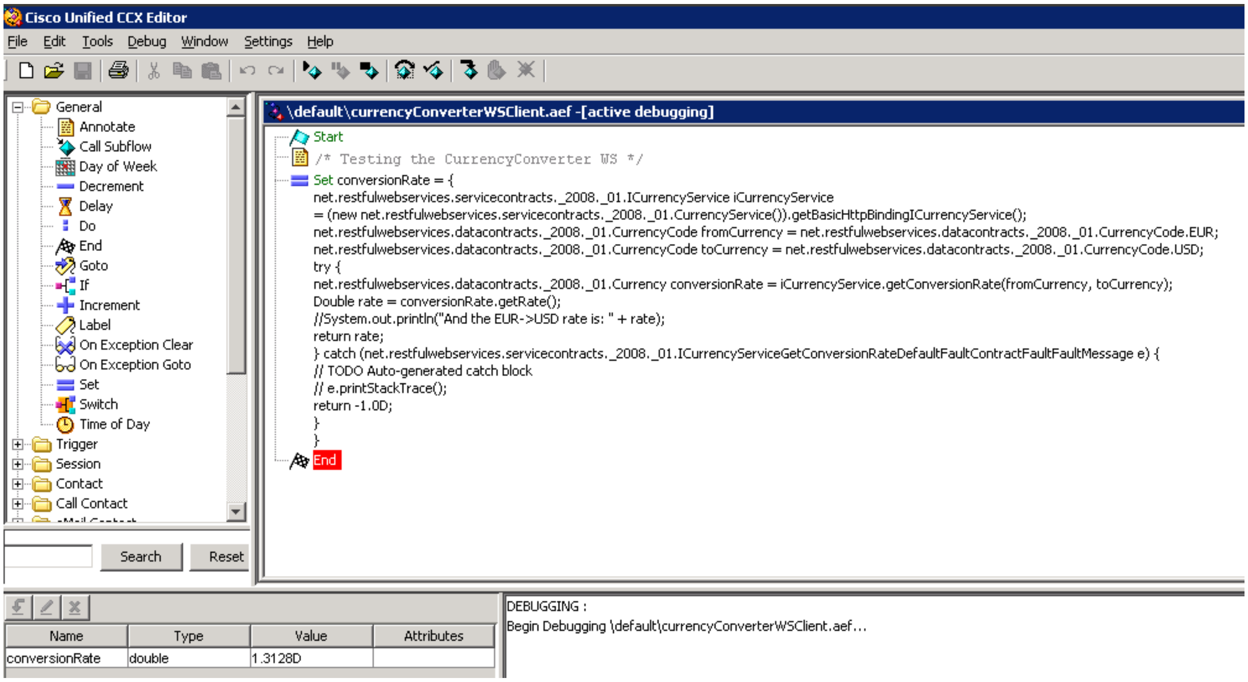
9. Let's create a script to test the WS functionality. One variable, of type `double`, will be needed. Name it `conversionRate`.

Then insert a Set step, where the return value would be the `conversionRate` variable, and the value is the following code block:

```
{
net.restfulwebservices.servicecontracts._2008._01.ICurrencyService
iCurrencyService
   =  (new
net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).
getBasicHttpBindingICurrencyService();
    net.restfulwebservices.datacontracts._2008._01.CurrencyCode
fromCurrency =
net.restfulwebservices.datacontracts._2008._01.CurrencyCode.EUR;
    net.restfulwebservices.datacontracts._2008._01.CurrencyCode
toCurrency =
net.restfulwebservices.datacontracts._2008._01.CurrencyCode.USD;
   try {
      net.restfulwebservices.datacontracts._2008._01.Currency
conversionRate = iCurrencyService.getConversionRate(fromCurrency,
toCurrency);
     Double rate = conversionRate.getRate();
     //System.out.println("And the EUR->USD rate is: " + rate);
     return rate;
    } catch
(net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGe
tConversionRateDefaultFaultContractFaultFaultMessage  e) {
     // TODO Auto-generated catch block
     // e.printStackTrace();
     return -1.0D;
   }
}
```

Remember, this was the program we tried before in Eclipse, with some modifications. For instance, if an exception is throw, we return -1 (or, more precisely, -1.0D) so the rest of the script knows there was an exception.

Take a deep breath and try to do a "Step Over" type debugging. If you see a non-negative value assigned to the conversionRate variable, congratulations, go out and celebrate.



```
Start
/* Testing the CurrencyConverter WS */
Set conversionRate = {
net.restfulwebservices.servicecontracts._2008._01.ICurrencyService iCurrencyService
= (new net.restfulwebservices.servicecontracts._2008._01.CurrencyService()).getBasicHttpBindingICurrencyService();
net.restfulwebservices.datacontracts._2008._01.CurrencyCode fromCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.EUR;
net.restfulwebservices.datacontracts._2008._01.CurrencyCode toCurrency = net.restfulwebservices.datacontracts._2008._01.CurrencyCode.USD;
try {
net.restfulwebservices.datacontracts._2008._01.Currency conversionRate = iCurrencyService.getConversionRate(fromCurrency, toCurrency);
Double rate = conversionRate.getRate();
//System.out.println("And the EUR->USD rate is: " + rate);
return rate;
} catch (net.restfulwebservices.servicecontracts._2008._01.ICurrencyServiceGetConversionRateDefaultFaultContractFaultFaultMessage e) {
// TODO Auto-generated catch block
// e.printStackTrace();
return -1.0D;
}
}
End
```

Name	Type	Value	Attributes
conversionRate	double	1.3128D	

```
DEBUGGING :
Begin Debugging \default\currencyConverterWSClient.aef...
```

If you get stuck or run into a problem:

- if it does not work in Eclipse, it won't work in UCCX, either;
- it's really necessary to reload the CCX engine;
- it takes a while to contact the WS server, after all, there's a lot going on behind the scenes;
- if you see an error message complaining about the class file version: remember, JDK 1.6 is supported (you probably have a different version).

Happy coding!

G.