

How to convert code from other languages



Huw Collingbourne finds out how to translate VB, C++ and Java into Delphi

Stealing other people's code is one of the great arts of programming. After all, if someone else has already solved a problem, it is a waste of time and effort to solve it all over again. But all too often, the code you want to steal is written in a different computer language.

This month we'll consider how to go about making sense of 'foreign' programming languages in general. We shall then tackle the translation of a small project from Java into Delphi. We have chosen the Java language because it not only uses almost the same syntax as C++ (and so is not quite the model of clarity that we Delphi programmers are used to!) but it also poses some unique problems of its own. For one thing, unlike Delphi, it is entirely object orientated and it has its own 'platform independent' libraries that are not closely bound to the Windows API.

If you can translate Java to Delphi, you should have no problems with Visual Basic and C++. Which means that, from now on, you'll be able to steal code from all the other *PC Plus* programming columns too!

Deadly syntax

The syntax and structures of another language may seem baffling at first sight. In its favour, it must be said that Visual Basic is fairly easy to understand. Its functions are preceded by the keyword *Function* and its procedures use the keyword *Sub*, short for 'Subroutine'. Variables are declared using the keywords *DIM* and *AS*:

```
Dim MyVariable As Integer
```

The Basic *FOR* loop is straightforward and uses a Delphi-like syntax but without the *BEGIN* and *END* delimiters. The end of the loop is indicated by the keyword *NEXT* followed by the loop's control variable as follows:

```
For i = 1 To 10  
  x = x + 1  
Next i
```

Armed with just this much knowledge, you shouldn't find it too difficult to start work on translating a moderately complex VB program into Delphi. The same, alas, cannot be said of C++ and Java! Neither of those languages provides a variable-declaration keyword. When

a variable is needed it is just declared and, optionally, initialised at the same time. The data type precedes the variable name and does not require a separating colon:

```
int x = 10;
```

No distinction is made between functions and procedures in Java or C++. If a value is to be returned, then the data type is placed before the function name, otherwise the keyword 'void' is used. The return value is indicated by the keyword 'return'. Where Delphi uses *BEGIN* and *END* delimiters, C++ and Java use curly brackets. Here, for example, a C++ function called *hi* takes a *CString* argument, concatenates it with "Hello world" and returns the result as a *CString* data type:

```
CString hi( CString s ) {  
  return "Hello world" + s ;  
}
```

There are a few other things you need to be beware of when converting C++ or Java code. For one thing, both of those languages are case sensitive. So, a variable called *X* is different from a variable called *x*. And the syntax of the 'for' loop will probably take some staring at if it's the first time you've seen it. Here is a C++/Java loop:

```
for( int i = 1; i <= 10; i++) {  
  x++;  
}
```

The loop header here has three parts. First it declares and initialises an *int* variable *i* to the value 1. The second part runs the loop only while the value of *i* remains less than or equal to 10. The third part increments *i* by 1 using the '++' operator at every turn through the loop. Here is the Delphi translation of this code (assuming that the integer variables *i* and *x* have previously been declared):

```
for i = 1 to 10 do  
  x := x + 1;
```

A fair wind for Java

Okay, that's enough theory for now. Let's turn to a practical example. We've written a simple Java applet (a program that can be embedded into a Web page) comprising a form containing a slider



SEE OUR **SUPERDISC**

PATH: \handson\files\delphi

bar, three buttons and a label. There are two code files in the project. The main unit, *JApplet1.java*, defines the visual layout of the applet and handles the mouse-click and slider-change events. The second file, *TestSrc.java*, contains three functions returning Strings.

At first sight, the main unit looks quite complex. However, if you read the comments you'll see that the bulk of the code has been automatically generated by the Java IDE, Visual Café. This code defines the visual interface. We can ignore it since Delphi's form designer looks after this when we drag controls on to a form.

The first piece of hand-written code starts here:

```
void JSlider1_stateChanged(javax.  
swing.event.ChangeEvent event)  
{  
  numberLabel.setText(String.val  
ueOf(JSlider1.getValue()));  
}
```

This handles the *Changed* event of the *JSlider* object. Look carefully at the code inside the brackets to see if you can work out what's going on here. Start at the innermost piece of code:

```
JSlider1.getValue()
```

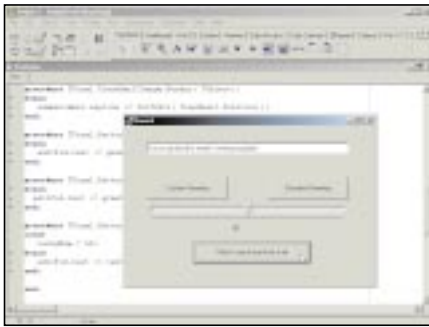
Obviously, this gets the current numerical value of the slider control. This value is then used by the code enclosing it:

```
String.valueOf()
```

In Java, *String* is a class and *valueOf()* is a class-bound procedure or 'method'. You don't need to know all this in order to deduce what's happening, though. It should be clear that the code is converting the numerical value of *JSlider1.getValue()* to a *String*. This *String* is then used by the piece of code at the outermost level:

```
numberLabel.setText()
```

Here *numberLabel* is the name of the label on the form. It's pretty obvious that *setText()* is a



The Delphi translation uses Windows controls such as TrackBar. The code translation makes allowances for the differing syntax of Delphi and the properties of the controls.

method of the label. So the entire block of code gets the value of the slider, converts it to a string and displays it in numberlabel.

Let's see how we go about translating this into Delphi. Load up the DelphiTest.dpr project. You'll see that we have recreated the Java form, using a Windows TrackBar object instead of a Java JSlider. Double-click the TrackBar to find the TrackBar1Change() event-handler. This is my Delphi translation of the Java code:

```
procedure TForm1.TrackBar1Change
(Sender:TObject);
begin
    numberlabel.caption := IntToStr(
    Trackbar1.Position );
end;
```

While the code looks very different from the Java original, it is, in essence, the same. First, it gets a numeric value, Trackbar1.Position, then it converts this to a string, IntToStr(), and finally it displays it in the label by assigning it to numberlabel.caption.

Now look at button3.ActionPerformed in the Java source and its Delphi equivalent, TForm1.Button3Click. These methods test the current slider or trackbar value when the button is clicked. Different messages are displayed in the edit box, depending on whether or not the value equals that of the constant, LuckyNum. There are two things to watch for here. First, Java uses the 'final' keyword to declare a constant:

```
final int LuckyNum = 50;
```

Second, notice the way in which the code calls the testnum() function. Ignoring the arguments to the function, this is the syntax used in each case:

```
editFld.text := testnum( );
// Delphi
textFld.setText( new TestSrc()
.testNum( ) ); // Java
```

Pay attention to the keyword 'new' in Java. As mentioned earlier, Java is fully object-orientated. If you look at the file in which the Java testNum() function occurs,

you'll see that it is embedded into a class with the same name as the code file, TestSrc. In Java, functions cannot exist outside of a class. This means that when one source file needs to use a function from another source file, it is not sufficient simply to import the file as is done at the top of JApplet1.java with an 'import' statement and in Test.pas with a 'uses' statement:

```
import TestSrc; // Java
uses TestSrc; // Delphi
```

In Java, it is also necessary to create an instance of the object whose method you want to use:

```
new TestSrc( )
```

We could, of course, slavishly copy the Java original and create a TestSrc object in Delphi. This would gain us nothing, however, other than an additional level of complexity. Finally, compare the methods in TestSrc.java with those in TestSrc.pas. The vital points to note here can be summarised as follows:

- In Java, the public keyword makes a method visible to other units. In Delphi, the methods must be declared in the interface section.
- In Java (and C++), brackets around an if test such as (num == winnum) are obligatory. In Delphi they are optional.
- In Java and C++, the == operator tests for equality and is comparable with Delphi's =. The C++/Java = operator is used for assignment

and is comparable with Delphi's :=. The + operator can be used to concatenate strings and the += operator concatenates and assigns in a single operation.

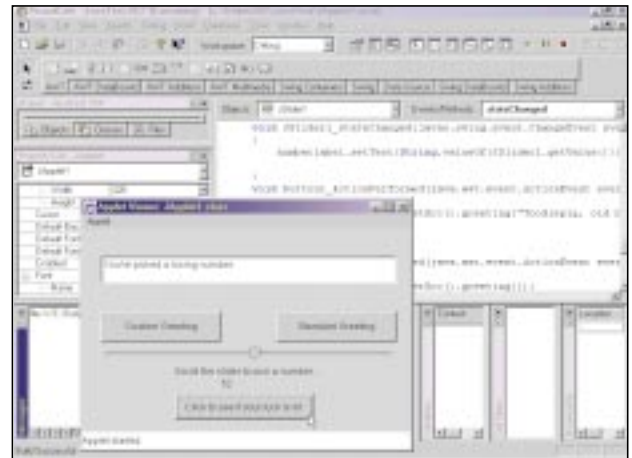
Finally, notice that the Java code uses two variations of a single method named greeting(), each of which is distinguished by differing arguments. In Delphi 3, this type of 'method overloading' is not permitted and we have had to declare two separate methods, greeting() and greeting2(). In Delphi 4 and above, method overloading is permissible, as long as the methods are declared with the 'overload' directive. You can see an example of this in the TestSrc5 unit in the \Delphi5Test directory.

Having wrestled with the elementary problems of code translation, the criminality continues next month. **PCP**

NEXT MONTH

We shall start work on a much more ambitious code-stealing project.

This project was programmed in Java using the Visual Café development environment. Java has its own non-Windows controls such as the Slider bar across the middle of the form.



VARIABLE VAGUARIES

True to type?

Take care when converting variables – particularly the strange strings used by Windows API calls

When you find a piece of foreign language code that you'd like to use, the first thing you need to do is to work out the equivalent data types. Often the translation of a type is self evident. For example, a Visual Basic Integer, and C++ or Java int are usually interchangeable with a Delphi Integer.

However, watch out for alternative 'flavours' of a given types. For example, Java has four integer types: byte – short, int and long – each of which has a range of values falling between a different maximum and minimum. If you look up

Delphi's help on Integer types, you will see that similar flavours of integer are provided in Delphi, too. If in doubt, use the type with the broadest range of values – so Delphi's Integer type can be substituted for any variety of Java integer. Well almost. It turns out that Java also has an Integer with a capital I. This is not a simple integer at all. It is a special type of object that provides methods for dealing with integers. In Java, be wary of anything with a capital initial!

Other exotic-looking data types occur in the Windows API documentation of Delphi's

on-line help. The API help is, inconveniently, written for C programmers. As a general rule, when a DWORD is documented, you can use a LongInt or an Integer. For UINT, use Integer and for LPCSTR substitute PChar. The API help also refers to some predefined structures such as SHFILEINFO. In most cases, the Delphi libraries provide an interface to these structures, often with the API name preceded by a T as in TSHFileinfo. In order to use these, you will normal need to add ShellAPI to the 'uses' section of the current unit.