Why Use BBj Custom Objects?

So you're a BBx programmer and someone is telling you that you should be using BBj Custom Objects -- or "worse" yet, the new code you get has custom objects in it.

What's the big deal about custom objects?

Collect Your Code

Well, let's start with a very simple and not very object-oriented example. Say you want to put all your functions into a class (a class is the blue-print for an object). You could use a class with static methods like this (methods are like functions):

```
class public Function
    method public static BBjString formatPhone(BBjString phone$)
        methodret phone$(1,3) + "." + phone$(4,3) + "." + phone$(7)
    methodend

    method public static BBjString someOtherFunction(...)
        ...
    methodend
classend
```

What you've done is create a class called Function. It's public which means that anyone can use it. It has a public, static method called formatPhone() that returns a BBjString (just a "normal" string) and takes one argument called phone$ which is also a BBjString. The phone$ gets a very simple format for this example. The someOtherFunction is just to remind you that a class can have many methods so you could put as many functions into it as you wanted. You would store this code in a file which you could call anything but let's call it "function.src".

Now to use your class you would do this:

```
rem Put the following line somewhere in your program, usually near the
top
use ::function.src::Function

...

phone$ = "5035551212"
pretty_phone$ = Function.formatPhone( phone$ )
print pretty_phone$; rem 503.555.1212
```

So is this better than simply putting def fnx code into your program? Yes, because simply by putting the "use" statement in any program, you now have all the functions in the Function class at your fingertips without cutting and pasting code. Better yet, if you improve formatPhone(), all your programs get it automatically.

But is it better than a called program? Not a lot. You could put all your functions into a called program with labels at separated "enter" lists and exits and get almost the same thing. One big difference (some would call it an advantage and some a disadvantage) is that you can have several call/enter arguments all of which can be going in or coming out or

both.

Use Objects

So let's look at a more typical way to use a class: to create objects. What is an object, anyway? Well, it's an "instance" of a class. If that didn't help, think of a class as a blue-print and an object as a house. But not just a cookie-cutter house, houses with different colors and number of rooms and things like that. Creating and using objects requires that we BBx programmers think in a different way about programming. We think less about what something does (procedures) and more about what is it (states and behaviors). The behaviors come from the methods as we've already seen. The states or properties come from the fields.

> ***Static methods and fields***
>
> *The methods in our first class were "static". This means they can be used without having an instance of the class. They act more like a subroutine or called program. Besides static methods, there's the concept of static fields. A static field exists only once in the variable space, for all instances of a class. Note that you can not access non-static fields from a static method.*

Without getting caught up in the code, lets look at a House class and then create an object from it.

```
class public House

    field public BBjString Color$
    field public BBjNumber Rooms

    method public void openDoor()
        ...
    methodend

    method public void closeWindow()
        ...
    methodend

classend
```

Our House class has two properties or fields: one for the color of the house and one for the number of rooms. It also has two behaviors: it can open a door and close a window. This is how you would use your House class:

```
use ::house.src::House

...

declare House myHouse!
declare House yourhouse!

myHouse! = new House()
myHouse!.setColor("green")
myHouse!.setRooms(3)
myHouse!.openDoor()
```

```
yourHouse! = new House()
yourHouse!.setColor("blue")
yourHouse!.setRooms(4)
yourHouse!.closeWindow()
```

You've seen the "use" statement before. The "declare" statement tells BBj what type of object this is. With x$ you know it's a string but with x! all you know is it's an object. What kind of object? The "use" statement says if you have an object of type House, you can find the class that describes it in the file house.src (it uses the normal prefix and current directory to find house.src). The "declare" statement says that the variables myHouse! and yourHouse! are House type objects. Now BBj knows what type of object they are (and it can "type-check" your program. More on that later.)

Now we need to get an "instance" of the House, in other words, your very own distinct house. You do this with the "new" keyword. The setColor() and setRooms() set the state of the house. myHouse! is green and has three rooms. yourHouse! is blue and has four room. Behavior is done using the methods. myHouse! has an open door. yourHouse! has a closed window.

Something More Real

So okay, the house object is cute, but what about a more real world example? What about customer addresses? (Normally, you'd probably make this part of a larger Customer class, but we're keeping it simple.)

```
class public CustomerAddress

rem --- Private fields; will need manual "getter" methods

    field private BBjString fullAddress$
    field private BBjString street$
    field private BBjString city$
      field private BBjString state$
    field private BBjString zipCode$

rem --- Constructor; use with the "new" keyword

    method public CustomerAddress(BBjString address_string$)
        #fullAddress$ = address_string$
        #parseString()
    methodend

rem --- Private method; only used inside this file

    method private void parseString()
        rem magically get the street, etc. from the address
        rem this code would normally be more complex
        #street$  = #fullAddress$(1, 100)
        #city$    = #fullAddress$(101, 50)
            #state$   = #fullAddress$(151, 2)
        #zipCode$ = #fullAddress$(153, 10)
```

```
    methodend

rem --- Public method; print formatted address

      method public void printFormattedAddress(BBjString name$)
            print name$
        print cvs(#street$, 2)
            print cvs(#city$, 2), ", ", #state$, "  ", #zipCode$
      methodend

rem --- Make fields "read-only" with "getters"

    method public BBjString getStreet()
            methodret #street$
      methodend

      method public BBjString getCity()
            methodret #city$
      methodend

      method public BBjString getState()
            methodret #state$
      methodend

      method public BBjString getZipCode()
            methodret #zipCode$
      methodend

classend
```

This class will create objects that take a long string with all the address information, parse the city, state, and zip code (very simplistically) and print a formatted address. This is more object oriented: the object "is" the address. Like our House object it has properties (fields, in this case the street, city, state and zip code) and behaviors (it can print the formatted address).

Now you have something a called program really can't do. In a single variable you're carrying around the data and actions on a particular customer address. With a call you'd have to always pass in the address or read it from the disk, and it would be for any customer address. An object will also keep its state. A call will always "start from zero", so to speak. An object can also "hide" things from the user they don't need to know, like the parseString() method. The user never needs to call this, so it's hidden (private). This may sound like an odd thing to a BBx program who is used to everything being open, but once the object is debugged, there is no reason to know what goes on in an object. That way you can change the way a method is implemented and everything still works as advertised.

Constructors

Most methods have a visibility (public, private, protected), the optional "static", and a return type. There's one method that has no return type and has the same name as the class. This is the "constructor". It's the method that gets called when you create an object with the "new" keyword. You use a constructor to initialize things once or do some setup. In this case we set a field and call an internal method.

If you don't create your own constructor, BBj uses the "default", which is as if you had written this:

```
method public CustomerAddress()
methodend
```

This is the kind of constructor we had in the House class.

Setters and Getters

In the House class we used the default setters and getters because those fields were public. For CustomerAddress, the fields are private so we have to write the setters and getters ourselves.

Setters and getters are very simple methods that do nothing except set or return (get) a field. The CustomerAddress class has this explicit getter:

```
method public BBjString getStreet()
        methodret #street$
  methodend
```

If we were going to make a setter for street it would look like this:

```
  method public void setStreet(BBjString street$)
        #street$ = street$
  methodend
```

(We'll explain the pound sign in a second.) Why would you want a getter without a setter? This is a way of making a field "read-only". The method parseString() sets the fields. You might not want the user to change that. But they can "get" the street with the explicit getter we wrote.

Scope

You put a pound sign (#) in front of field and method names that are in the class. That's because unlike most BBx code, fields and methods aren't "global". We BBx programmers are used to variables being "seen" from anywhere. In classes, variables are seen only within a method. This is called a local variable (where you can see a variable is a simple way of defining the scope of the variable.) You use the pound sign to signify you are accessing a field or method inside the class but outside of the method you're in. (It's a little more complex than that, but that's good enough for most cases.)

So in the setStreet() setter we wrote in the section above, we have a local variable "street$" and a global field "#street$". Without the pound sign, we couldn't tell which "street$" we were accessing.

Parameters, Arguments, and Type

Let's think about the old BBx DEF FNx for a second. We might write:

```
def fnformat_date$(q$)
      if len(q$) > 7 then
            return q$(5,2) + "/" + q$(7,2) + "/" + q$(1,4)
```

```
        else
              return q$
        endif
fnend
```

You might use this function like this:

```
print fnformat_date$("20100131"); rem 01/31/2010
```

The q$ in the function definition is called a parameter and the string in the print statement ("20100131") is called an argument. BBx is loosely typed. This means that the types of variables are not strictly enforced. We can see from function definition that q$ is a string and you will get a run-time error if you use something that does not evaluate to the string as an argument.

```
print fnformat_date$(some_number)
```

But the program will still compile. You have to actually run into the line of code for the error to be apparent. We can also tell the function returns a string by the dollar sign ($) after the function name, so this line won't compile:

```
x = fnformat_date$("20100131")
```

But the following code will compile just fine. It is only when you run it that you get an error:

```
x$ = fnbad_date$(q$)

def fnbad_date$(q$)
    return 123
fnend
```

With BBj classes all these holes are plugged. Everything must be typed; arguments must match parameters and return types are checked. You can check that your program is "type safe" by using the BBjCpl program with the -t flag for type checking and the -W flag to warn about undeclared variables (more about them later).

So looking at our street getter...

```
method public BBjString getStreet()
        methodret #street$
  methodend
```

...the method declares that it returns a BBjString (this is the type of the variable x$). With our street setter...

```
method public void setStreet(BBjString street$)
        #street$ = street$
  methodend
```

...the return type is "void" (meaning it returns nothing) and the first parameter is type BBjString. The arguments must match the parameters and the return types are checked. So BBj classes are strongly typed. It makes it a little harder to program but the programs are much safer and less likely to throw run-time errors.

Declare and Cast

In BBx, this will get a compile time error:

```
x = x$
```

So we know to do this:

```
x = num(x$)
```

We don't call it this but that's a "cast". A cast is a way of changing the type. Of course, just as x$ in this example must hold something numeric, a cast only works if the type is "castable". The BBj way of doing the above would be:

```
x = cast(BBjNumber, x$)
```

The classic BBx variables have the following types:

- String (x$), BBjString
- Integer (x%), BBjInt
- Number (x), BBjNumber

These will get you a long way. But it would be impossible to have some kind of character to designate every type of object. x! is an object, but what kind of object? How can we type check it? You would use a declare statement:

```
declare BBjVector x!
```

This states that x! is a type of BBjVector. Now not only is the type checked, but the methods you can call on it are checked too:

```
rem the method getVector() must return a BBjVector
x! = someObject!.getVector()

rem this will fail, there is no such method in a BBjVector
x!.doSomethingWeird()
```

So when do we use a declare statement? You use it to set the type of the variable. In classes, some things are already typed. Fields have a type; methods have a return type and their parameters are typed. Local variables are not typed so these often need to be declared. Let's create a new method for our CustomerAddress class that returns a formatted address in a BBjVector.

```
method public BBjVector getFormattedAddress(BBjString name$)
      declare BBjVector address!
      address! = BBjAPI().makeVector()
      address!.addItem(name$)
      address!.addItem(#street$)
      address!.addItem(#city$)
      address!.addItem(#state$)
      address!.addItem(#zipCode$)
      methodret address!
methodend
```

You declare address! because you want to use it as a BBjVector and BBjAPI().makeVector() returns a BBjVector. The type checker will make sure that there

is a method called addItem() and that the argument matches the parameter. To use this method in CustomerAddress you would do something like this:

```
use ::cust_addess.src::CustomerAddress
...
declare CustomerAddress ca!
declare BBjVector address!

ca! = new CustomerAddress(unformatedString$)
address! = ca!.getFormattedAddress("George Washington")
city$ = cast( BBjString, address!.getItem(2) )
rem ...or...
city$ = str( address!.getItem(2) )
```

Why the cast (or str() function) on address!.getItem(2)? If you look up the getItem() method of a BBjVector, you find it returns an Object. This allows you to put just about anything into a BBjVector, but it means you're not sure what type of object it returns. Therefore, knowing that the city is a string, you cast it to BBjString.

What's Next?

At this point it's probably a good time to read David Wallwork's pieces BBj Custom Objects Tutorial and the Type Checker Overview. There you'll learn about how you can extend objects (use class House to make a new class RanchHouse), implement interfaces (make a group of objects all Sortable) use GUIs in objects, throw errors, and more.

Just in is Adam Hawthorne's Object-Oriented Syntax Enhancements where you'll learn about inner classes, Java interfaces, type conversions and more.

Appendix A

Here's a simple way to get a type checker (in Windows) using a batch file and a shortcut to your Desktop:

```
@echo off
echo Type checking...

rem --- Usage:
rem --- Save this to "type_checker.bat"
rem --- Make a shortcut to it on your desktop
rem --- (In the file explorer, right-click and "send to desktop")
rem --- Then drag and drop source files onto the shortcut

rem --- Change these to match your system
set BAR_HOME=C:\devel\main\barista\
set BBJCPL=C:\Program Files\basis\bin\BBjCpl.exe
set BAR_CONFIG=C:\devel\main\barista\sys\config\enu\barista.cfg

rem --- Real work goes on here
cd "%BAR_HOME%"
"%BBJCPL%" -X -N -t -W -c"%BAR_CONFIG%" %1
pause
```

To turn on type checking in the BASIS IDE, go to Tools -> Options, and from here expand the Building node and click on BBj Compiler Settings. On the right-hand side, expand

the Expert node. There you can check Perform Type Checking and Show Type Check Warnings. You will probably also want to enter the path to you Barista configuration file in the field Select Prefix Directory Source.

Appendix B

Here's a full listing of the Customer Address object. It has code to test it at the end so you can just run the code and the tests will start. This is often a good way to include self-tests in classes.

```
rem --- Class to teach about objects

      class public CustomerAddress

      rem --- Private fields; will need manual "getter" methods

          field private BBjString fullAddress$
          field private BBjString street$
          field private BBjString city$
          field private BBjString state$
          field private BBjString zipCode$

      rem --- Constructor; use with the "new" keyword

          method public CustomerAddress(BBjString address_string$)
              #fullAddress$ = address_string$
              #parseString()
          methodend

      rem --- Private method; only used inside this file

          method private void parseString()
              rem magically get the street, etc. from the address
              rem this code would normally be more complex
              #street$  = #fullAddress$(1, 100)
              #city$    = #fullAddress$(101, 50)
              #state$   = #fullAddress$(151, 2)
              #zipCode$ = #fullAddress$(153, 10)
          methodend

      rem --- Public method; print formatted address

          method public void printFormattedAddress(BBjString name$)
              print name$
              print cvs(#street$, 2)
              print cvs(#city$, 2), ", ", #state$, "  ", #zipCode$
          methodend

      rem --- Public method; return a BBjVector of the address parts

          method public BBjVector getFormattedAddress(BBjString name$)
              declare BBjVector address!
              address! = BBjAPI().makeVector()
              address!.addItem(name$)
              address!.addItem(#street$)
```

```
            address!.addItem(#city$)
            address!.addItem(#state$)
            address!.addItem(#zipCode$)
            methodret address!
        methodend

    rem --- Make fields "read-only" with "getters"

        method public BBjString getStreet()
            methodret #street$
        methodend

        method public BBjString getCity()
            methodret #city$
        methodend

        method public BBjString getState()
            methodret #state$
        methodend

        method public BBjString getZipCode()
            methodret #zipCode$
        methodend

    classend

rem --- Testing

    declare CustomerAddress ca!
    declare BBjVector address!

    dim unformatedString$(162)
    unformatedString$(1,100)  = "123 NW Main Street"
    unformatedString$(101,50) = "Walnut Creek"
    unformatedString$(151,2)  = "CA"
    unformatedString$(153,10) = "95945"

    ca! = new CustomerAddress(unformatedString$)
    address! = ca!.getFormattedAddress("George Washington")
    city$ = cast( BBjString, address!.getItem(2) )
    rem ...or...
    city$ = str( address!.getItem(2) )
    print "City: ", city$

    print 'lf',"Formatted Address:"
    ca!.printFormattedAddress("George Washington")

    end
```