BBj Custom Objects Tutorial

June 1, 2006

By

David Wallwork – Software Architect, BASIS International Ltd.

This tutorial introduces custom objects, which bring the power of object-oriented programming to the BBj language.

Contents

BBj Custom Objects Tutorial

Introduction to BBj Custom Classes

Beginning with version 6.0, BBj® supports custom classes, which enable the BBj programmer to write object-oriented code. For example, the following program writes "hello world" to the console:

```
0010 rem speaker.src
0020 declare Speaker Speaker!
0030 Speaker! = new Speaker()
0040 Speaker!.speak()
0050 stop
0060 class public Speaker
0070   method public void speak()
0080     print "hello world"
0090   methodend
0100 classend
```

This paper provides a short tutorial on writing object-oriented code in BBj.

A BBj custom class consists of a collection of fields and methods that are defined by the BBj programmer. A custom object is an instance of a custom class. Custom objects can be used in BBj code in the same way that Java objects can be used.

This tutorial will discuss several examples without formally defining the specific syntax of each statement. For more information, refer to the online documentation for the **class**, **classend**, **field**, **interface**, **interfaceend**, **method**, **methodend**, and **methodret** verbs.

This section will provide some general information about custom classes and custom objects. The remaining sections of this tutorial provide samples that demonstrate how custom objects work within BBj code. Throughout the sample code we use the **declare** verb. The **declare** verb helps the developer find type-check problems that might result in runtime errors.

The definition of a custom class can be contained within a file that also contains other BBj program code. As the interpreter moves through a program that contains a custom class definition it will skip over the custom class definition (**`class`** through **`classend`**) in the same way that it will skip over the definition of a user-defined function (**`def fn`** through **`fnend`**).

An instance of a custom class is created using the **new** operator just as an instance of a Java object is created using the **new** operator. An instance of a custom class is referred to as a custom object.

A custom class **B** can extend another custom class **A**. In this case, we refer to custom class **A** as the superclass of custom class **B** and we refer to custom class **B** as a subclass of custom class **A**.

A custom class method defines a variable scope. The only variables that are shared between the code within a custom class method and the code that called the custom class method are those

variables that have been passed in the parameter list of the custom class method. The fields of a custom object are accessible within all methods of the custom object.

The methods of a custom object are invoked by using the dot notation that is common to most object-oriented languages. A method of a custom class can be public, protected or private. These protection levels are similar to the protection levels in other object-oriented languages.

The fields of a custom object can be public or private. The fields of a custom object cannot be accessed using the dot notation that is common to other object-oriented languages. Most code accesses the fields of a custom object using *accessor* methods (see the following example). BBj automatically generates a pair of accessor (get/set) methods for every field. These accessor methods have the protection level that is declared on the fields being accessed.

The special symbol **#** is used within the code of a custom object method to access the fields and the methods of that custom object. Within a method, code uses **`#fieldName`** to access a field whose name is **`fieldName`** and **`#methodName`** to access a method whose name is **`methodName`**.

Every custom object has an implicit field with the name **`this!`**. Code that resides within a non-static method of a custom object can access this implicit field using the notation **`#this!.`** This implicit field is not accessible to code other than the code within a non-static method of the custom object. If the custom class that defines a custom object has a superclass, then there is a second implicit field having the name **`super!`** that can be accessed within any non-static method using the notation **`#super!`**. The following sections will provide samples for using **`#this!`** and **`#super!`**.

The fields of a custom object are strongly typed. The type of each field is required as part of the declaration of the field. Similarly, the parameters and return types of all custom object methods are strongly typed and the return types are required as part of the declaration of the methods. With this general information, let's look at some sample code.

A Simple Custom Object: Writing a Check

In this section – We introduce our first custom object. We write a check to Jenny.

This simple program creates an instance of a Check at line **`0030`** and then prints out the payee's name and the amount of the Check at line **`0040`**. The definition of the Check is found at lines **`0060`** through **`0160`**.

```
0010 rem ' payable_01.src
0020 declare Check check!
0030 check! = new Check("Jenny Jones", 50.44)
0040 print check!.getPayee()," was paid ",check!.getAmount()
0050 rem ' flow control skips over the class definition
0060 class public Check
0070   field public BBjString Payee$
0080   field public BBjNumber Amount
0090   method public Check(BBjString Payee$, BBjNumber Amount)
0100     #Payee$ = Payee$
0110     #Amount = Amount
```

```
0120   methodend
0130   method public void setAmount(BBjNumber Amount)
0140     print "Amount of a Check can not be set after it is written"
0150   methodend
0160 classend
0170 print " we will now change the payee of the check"
0180 check!.setPayee("Joseph Jones")
0190 print check!.getPayee()," was paid ", check!.getAmount()
0200 print " and we will attempt to change the amount"
0210 check!.setAmount(43223.17)
0220 print check!.getPayee()," was paid ", check!.getAmount()
```

Some things to notice

- Method names and field names are case sensitive. In the statement, **`#Payee$ = Payee$`**, the field reference **`#Payee$`** is case-sensitive, but **`Payee$`**, a traditional BBx® string variable, is *not* case-sensitive.
- BBjString fields and parameter names follow string variable naming rules (x$ or x!).
- BBjInt fields and parameter names follow integer variable naming rules (x% or x!).
- BBjNumber fields and parameter names follow numeric variable naming rules (x or x!).
- All other field and parameter names follow object naming rules (x!).
- The class types of all fields are defined in the **field** statement. The class types of all method parameters are defined in the **method** statement. By using the **declare** statement, the program can also define the class type of local variables. When a class type has been defined for a variable BBj will enforce type safety for that variable.
- Because **`Payee$`** and **`Amount`** are declared as "public," the accessors for these fields are also public and are equivalent to:

```
method public void setAmount(BBjNumber Amount)
    #Amount = Amount
methodend

method public BBjNumber getAmount()
    methodret #Amount
methodend

method public void setPayee(BBjString Payee$)
    #Payee$ = Payee$
methodend

method public BBjString getPayee()
    methodret #Payee$
methodend
```

- The developer can override the accessors to define more specific actions.

Private Fields: Derived Classes, Protected and Private Fields

In this section – We create a new class **PayrollCheck** that is a subclass of **Check.**

```
0010 rem ' payable_02.src
0020 class public Check
0030   field private BBjString Payee$
0040   field private BBjNumber Amount
0050   field private BBjNumber NextCheckNumber = 1
0060   field protected BBjNumber CheckNumber
0070   method protected Check(BBjString Payee$, BBjNumber Amount)
```

```
0080     #Payee$ = Payee$
0090     #Amount = Amount
0100     #NextCheckNumber = #NextCheckNumber + 1
0110   methodend
0120   method public BBjNumber getAmount()
0130     methodret #Amount
0140   methodend
0150   method public BBjString getPayee()
0160     methodret #Payee$
0170   methodend
0180 classend
0190 class public PayrollCheck extends Check
0200   field private BBjNumber NetAmount
0210   field private BBjNumber TaxAmount
0220   method public PayrollCheck(BBjString Name$, BBjNumber NetAmount, BBjNumber TaxAmount)
0230     #super!(Name$, NetAmount - TaxAmount)
0240     #NetAmount = NetAmount
0250     #TaxAmount = TaxAmount
0260   methodend
0270   method public BBjNumber getNetAmount()
0280     methodret #NetAmount
0290   methodend
0300   method public BBjNumber getTaxAmount()
0310     methodret #TaxAmount
0320   methodend
0330 classend
0340 declare Check paycheck!
0350 paycheck! = new PayrollCheck("Joe Diamond", 278.35, 73.22)
0360 print paycheck!
```

Some things to notice

- The **methodret** verb is used to return a value from a custom object method. If the method has been declared to return void then there must be no expression following the **methodret**. If the method has been declared to have a non-void return type then there must be an expression following **methodret** and that expression must evaluate to the declared return type of the method.
- If a method has been declared to return void, then there is an implicit **methodret** prior to the **methodend** statement. This means that a method will return to the caller when it reaches the **methodend** statement. If the method has been declared to have a non-void return type, then there is no implicit **methodret**. If the code within a method runs to the **methodend** statement, an error will be generated when the **methodend** statement is executed.
- The keyword **extends** is used to create a derived class.
- Methods declared **protected** can be accessed within derived classes, but methods that declared **private** cannot.
- Fields can have an optional initialization expression. If there is no initialization expression then the initial value of a field is dependent on its type. A field of type BBjString will be initialized to contain an empty string (""). A field of type BBjNumber or of type BBjInt will have an initial value of zero. All other fields will be initialized to null.
- The definition of a custom class must begin with the **class** statement and end with the **classend** statement.
- Every method must begin with a **method** declaration and end with **methodend**.

- Custom object fields that contain the traditional BBx strings are declared as having type BBjString. Custom objects that contain traditional BBx numbers are declared as BBjNumbers or as BBjInts.
- The variable **paycheck!** is declared as a **Check** but has been assigned a value that is a **PayrollCheck**. This is a valid assignment since a **PayrollCheck** is by declaration a **Check**. But since **paycheck!** is declared as a **Check**, only the methods of a **Check** can be called on **paycheck!**.

The USE statement: USE and STATIC

In this section – We split a program into two files and introduce the **use** directive. We demonstrate the use of static methods.

```
0010 REM ' accounting_03.src
0020 USE ::payable_03.src::Check
0030 USE ::payable_03.src::PayrollCheck
0040 declare Check aCheck!
0050 declare PayrollCheck bCheck!
0060 declare Check cCheck!
0070 aCheck! = new PayrollCheck("Jenny Jones", 507.44, 189.49)
0080 print aCheck!.getNextCheckNumber()
0090 bCheck! = new PayrollCheck("Timmy Malone", 50.44, 189.49)
0100 print aCheck!.getNextCheckNumber()
0110 print bCheck!.getNextCheckNumber()
0120 print Check.getNextCheckNumber()
0130 print PayrollCheck.getNextCheckNumber()
0140 print "we can create a PayrollCheck but we can not create a Check"
0150 cCheck! = new Check("Jenny Jones", 54884.99)
```

```
0010 rem ' payable_03.src
0020 class public Check
0030   field private BBjString Payee$
0040   field private BBjNumber Amount
0050   field private static BBjNumber NextCheckNumber = 1
0060   field protected BBjNumber CheckNumber
0070   method protected Check(BBjString Payee$, BBjNumber Amount)
0080     #Payee$ = Payee$
0090     #Amount = Amount
0100     #NextCheckNumber = #NextCheckNumber + 1
0110   methodend
0120   method public BBjNumber getAmount()
0130     methodret #Amount
0140   methodend
0150   method public BBjString getPayee()
0160     methodret #Payee$
0170   methodend
0180   method public static BBjNumber getNextCheckNumber()
0190     methodret #NextCheckNumber
0200   methodend
0210 classend
0220 class public PayrollCheck extends Check
0230   field private BBjNumber NetAmount
0240   field private BBjNumber TaxAmount
0250   method public PayrollCheck(BBjString Name$, BBjNumber NetAmount, BBjNumber TaxAmount)
0260     #super!(Name$, NetAmount - TaxAmount)
0270     #NetAmount = NetAmount
0280     #TaxAmount = TaxAmount
0290   methodend
0300   method public BBjNumber getNetAmount()
0310     methodret #NetAmount
0320   methodend
0330   method public BBjNumber getTaxAmount()
0340     methodret #TaxAmount
```

```
0350   methodend
0360 classend
```

Some things to notice

- The **use** statement tells the program where to find the definition of a custom class that appears in another file. The program must contain a distinct **`use`** statement for each external custom class that it is going to use.
- The **use** statement consists of two parts. The first part is a filename surrounded by **::** symbols. The second part is the name of a custom class in that file.
- If the filename contained in the **use** statement is not an absolute filename, then BBj will resolve the filename using normal BBj search rules (current working directory and prefix).
- BBj custom objects support static fields and static methods. In this example, the methods **aCheck!.getNextCheckNumber()** and **bCheck!.getNextCheckNumber()** return the same value. As more checks are created, the value returned by **getNextCheckNumber()** will change.
- The method **getNextCheckNumber()** can be called on the custom class itself using the name of the custom class. So **Check.getNextCheckNumber()** and **PayrollCheck.getNextCheckNumber()** return the same value as **aCheck!.getNextCheckNumber()** and **bCheck!.getNextCheckNumber()**.
- A static method cannot access non-static fields.
- Static fields and static methods are static within a BBj session. If **`accounting_03.src`** is run in two distinct BBj sessions, then each session will start with check number 1; the two sessions will increment their check numbers independently.
- A static field persists throughout the lifetime of a BBj session.
- The keyword **extends** indicates that the class **PayrollCheck** is a subclass of the class **Check**.
- A method that is **private** can only be accessed by the declaring class. It cannot be accessed by derived classes and it cannot be accessed by code outside the class.
- A protected method can be accessed by a derived class whether that derived class is defined in the same file or in a separate file.
- In **`accounting_03.src`** the local variable **aCheck!** is declared as having type **Check**. The local variable **bCheck!** is declared as having type **PayrollCheck**. Because **PayrollCheck** extends **Check**, we can assign a new **PayrollCheck** to **aCheck!** or to **bCheck!**.
- The protected constructor of **Check** cannot be invoked from the file **`accounting_03.src`**.

Error Handling in Custom Object: seterr, setopts, and THROW

In this section – We demonstrate the use of **seterr** within a custom object method and introduce the **throw** verb.

```
0010 rem errorHandling.src
0020 seterr errorHandler
0030 declare BBjString options$
0040 declare BBjString filename$
0050 declare Foo foo!
0060 input "allow debug in public program?  yes=1/no=0 ",allow
0070 options$ = opts
0080 if allow then
0090    options$(1,1) = ior(options$(1,1),$08$)
0100    print "program will stop on error"
0110 else
0120    options$(1,1) = and(options$(1,1),$f7$)
0130    print "program will go to error handler on error"
0140 endif
0150 setopts options$
0160 filename$ = "ljl;kj;"
0170 foo! = new Foo()
0180 foo!.openFile(filename$)
0190 end
0200 class public Foo
0210   method public void openFile(BBjString filename$)
0220     print "Foo.openFile() opening file: ",filename$
0230     seterr fileOpenError
0240     open (unt)filename$
0250     print "Succeeded opening ",filename$
0260     methodret
0270 fileOpenError:
0280     print "entering error handler for Foo.fileOpenError"
0290     throw "Foo.openFile() could not open file: "+filename$,err
0300   methodend
0310   method public void bar()
0320     throw "throw in bar",22
0330   methodend
0340 classend
0350 errorHandler:
0360   print "entering error handler for program"
0370   print "received error:",errmes(-1)
0380   fileName$ = pgm(-1)
0390   retry
```

Some things to notice

- Error handling within a method of a custom object is similar to error handling in a called program. If the **setopts** bit that allows escape in a called program has been set, then an error in a called program will cause the program to drop to console. If that bit has not been set then an error in the program will cause control to return to the calling program. Setting that bit affects the behavior of custom object methods in the same way. If it is set then an error in the method will cause the program to drop to console. If it is not set then an error in a method will cause control to return to the calling program.
- The **seterr** verb is available within a custom object method. The effect of a **seterr** within a method is similar to the effect of a **seterr** in a called program. The **seterr** remains in affect until the method returns.
- By using **throw**, a program can cause an error to occur programmatically. This allows the programmer to specify the error number as well as the error message.

Polymorphic methods: Paying Salaried vs. Commissioned Employees

In this section – We introduce two more derived classes, **SalariedEmployee** and **CommissionedEmployee**, as well as a **PayMaster** class. **PayMaster** has two methods for paying employees; one for paying a salaried employee and another for a commissioned employee. The caller with an employee in hand can call **PayMaster.pay()** to generate a check for either type of employee. **PayMaster** then manages the differences between paying a salaried employee and a commissioned employee.

```
0010 REM ' accounting.src
0020 use ::employee.src::Employee
0030 use ::employee.src::EmployeeIF
0040 use ::employee.src::SalariedEmployee
0050 use ::employee.src::CommissionedEmployee
0060 use ::payable_04.src::PayMaster
0070 use ::payable_04.src::Check
0080 use ::payable_04.src::PayrollCheck
0090 declare PayMaster PayMaster!
0100 PayMaster! = new PayMaster()
0110 declare CommissionedEmployee James!
0120 James! = new CommissionedEmployee("James Dean", 0.16)
0130 James!.addSale(5219)
0140 James!.addSale(811)
0150 declare SalariedEmployee Bob!
0160 Bob!= new SalariedEmployee("Bob Wills", 50000, 2)
0170 Bob!.setRetirementDeduction(225)
0180 declare EmployeeIF Employee!
0190 Employee! = James!
0200 declare Check checkForJames!
0210 checkForJames! = Employee!.getPaid(PayMaster!)
0220 Employee! = Bob!
0230 declare Check checkForBob!
0240 checkForBob! = Employee!.getPaid(PayMaster!)
0250 print "James received check #", checkForJames!.getCheckNumber(),
0260 print " for amount of ", checkForJames!.getAmount()
0270 print "Bob received check #", checkForBob!.getCheckNumber(),
0280 print " for net amount of ", checkForBob!.getAmount()
0290 Employee! = James!
0300 print "Does James receive commission?",PayMaster!.receivesCommission(Employee!)
0310 Employee! = Bob!
0320 print "Does Bob receive commission?",PayMaster!.receivesCommission(Employee!)
```

```
0010 rem ' payable_04.src
0020 use ::employee.src::Employee
0030 use ::employee.src::EmployeeIF
0040 use ::employee.src::SalariedEmployee
0050 use ::employee.src::CommissionedEmployee
0060 rem ' ============================================================
0070 class public PayMaster
0080   field public BBjString Name$
0090   field public BBjNumber BonusThreshold = 2000
0100   field private BBjNumber PayPeriodsPerYear = 24
0110   field private BBjNumber StandardDeduction = 200
0120   field private BBjNumber TaxRate = 0.30
0130   method public Check pay(SalariedEmployee Employee!)
0140     Name$ = Employee!.getName()
0150     net = round((Employee!.getSalary()/#PayPeriodsPerYear),2)
0160     ira = Employee!.getRetirementDeduction()
0170     deductions = round(Employee!.getDependents()*#StandardDeduction+ira,2)
0180     tax = round(#TaxRate*(net-ira-deductions),2)
0190     methodret new PayrollCheck(Name$, net, tax)
0200   methodend
0210   method public Check pay(CommissionedEmployee Employee!)
0220     sales = Employee!.getUnpaidSales()
0230     net = round(Employee!.getCommission()*sales,2)
0240     if sales >= #BonusThreshold
0250        net = net + 100
```

```
0260     endif
0270     Employee!.setUnpaidSales(0)
0280     Name$ = Employee!.getName()
0290     methodret new Check(Name$, net)
0300   methodend
0310   method public BBjNumber receivesCommission(EmployeeIF Employee!)
0320     cast(CommissionedEmployee,Employee!,err=not)
0330     methodret 1
0340   not:
0350     methodret 0
0360   methodend
0370 classend
0380 rem ' ============================================================
0390 class public Check
0400   field protected static BBjNumber NextCheckNumber = 1
0410   field private BBjNumber CheckNumber
0420   field public BBjString Payee$
0430   field protected BBjNumber Amount
0440   method public Check(BBjString Payee$, BBjNumber Amount)
0450     #Payee$ = Payee$
0460     #Amount = Amount
0470     #CheckNumber = #NextCheckNumber
0480     #NextCheckNumber = #NextCheckNumber + 1
0490   methodend
0500   method public BBjNumber getAmount()
0510     methodret #Amount
0520   methodend
0530   method public BBjNumber getCheckNumber()
0540     methodret #CheckNumber
0550   methodend
0560   method public static BBjNumber getCheckCount()
0570     methodret #NextCheckNumber
0580   methodend
0590 classend
0600 rem ' ============================================================
0610 class public PayrollCheck extends Check
0620   field protected BBjNumber NetAmount
0630   field protected BBjNumber TaxAmount
0640   method public PayrollCheck(BBjString Name$, BBjNumber NetAmount, BBjNumber TaxAmount)
0650     #super!(Name$, NetAmount-TaxAmount)
0660     #NetAmount = NetAmount
0670     #TaxAmount = TaxAmount
0680   methodend
0690 classend
```

```
0010 rem ' ============================================================
0020 rem ' employee.src
0030 rem ' ============================================================
0040 use ::payable_04.src::PayMaster
0050 use ::payable_04.src::Check
0060 class public Employee implements EmployeeIF
0070   field public BBjString Name$
0080   method protected Employee(BBjString Name$)
0090      #Name$ = Name$
0100   methodend
0110   method public Check getPaid(PayMaster PayMaster!)
0120      throw "Base class Employee.getPaid() should never be invoked",17
0130   methodend
0140 classend
0150 rem ' ============================================================
0160 class public SalariedEmployee extends Employee implements EmployeeIF
0170   field private BBjNumber Salary
0180   field public BBjNumber Dependents
0190   field public BBjNumber RetirementDeduction
0200   method public SalariedEmployee(BBjString Name$, BBjNumber Salary, BBjNumber dependents)
0210     #super!(Name$)
0220     #Salary = Salary
0230     #Dependents = Dependents
0240     #RetirementDeduction = 0
0250   methodend
```

```
0260   method public BBjNumber getSalary()
0270     methodret #Salary
0280   methodend
0290   method public Check getPaid(PayMaster PayMaster!)
0300     methodret PayMaster!.pay(#this!)
0310   methodend
0320 classend
0330 rem ' ============================================================
0340 class public CommissionedEmployee extends Employee implements EmployeeIF
0350   field public BBjNumber Commission
0360   field public BBjNumber UnpaidSales
0370   method public CommissionedEmployee (BBjString Name$, BBjNumber Commission)
0380     #super!(Name$)
0390     #Commission = Commission
0400     #UnpaidSales = 0
0410   methodend
0420   method public void addSale(BBjNumber Amount)
0430     #UnpaidSales = #UnpaidSales + Amount
0440   methodend
0450   method public Check getPaid(PayMaster PayMaster!)
0460     methodret PayMaster!.pay(#this!)
0470   methodend
0480 classend
0490 rem ' ============================================================
0500 interface public EmployeeIF
0510   method public Check getPaid(PayMaster PayMaster!)
0520   method public BBjString getName()
0530 interfaceend
```

Some things to notice

- In **`accounting_04.src`** we can pay either employee by calling **PayMaster!.pay()**. Passing a **SalariedEmployee** results in a call to the method **pay(SalariedEmployee)** and also withholds taxes. When passing a **CommissionedEmployee**, the method **pay(CommissionedEmployee)** is called and no taxes are withheld.
- Because **nextCheckNumber()** is static in the base class **Check**, the check numbers for **Checks** and **PayrollChecks** are correctly sequenced.

A GUI Example: Registering Custom Object Methods as Callbacks, Callbacks Within a Method Body

In this section – We create a simple dialog box that can be called from character code. This program demonstrates the registering of custom object methods as callbacks and also demonstrates the use of traditional callbacks (using labels) within a custom object method.

```
0010 rem dialog.src
0020 open (unt)"X0"
0030 declare Dialog dialog!
0040 declare Reporter reporter!
0050 dialog! = new Dialog()
0060 print "About to show dialog"
0070 dialog!.init()
0080 reporter! = new Reporter()
0090 dialog!.registerEvents(reporter!)
0100 dialog!.show()
0110 print "Button was pushed",dialog!.getClickCount()," times."
0120 escape
0130 print "show it again"
0140 dialog!.init()
0150 dialog!.registerEvents(reporter!)
0160 dialog!.show()
0170 print "finished"
0180 stop
```

```
0190 class public Dialog
0200   field private BBjWindow Window!
0210   field private BBjButton Button!
0220   field private BBjStaticText Text!
0230   field private BBjStaticText Text2!
0240   field private BBjStaticText Counter!
0250   field private BBjInputE InputE!
0260   field private BBjSysGui SysGui!
0270   field private BBjNumber Active = 0
0280   field public BBjNumber ClickCount = 0
0290   method public Dialog()
0300     #SysGui! = bbjapi().getSysGui()
0310   methodend
0320   method public void init()
0330     #Window! = #SysGui!.addWindow(100,10,10,200,220,"Test Dialog")
0340     #Button! = #Window!.addButton(200,50,40,80,40,"Push me")
0350     #Text! = #Window!.addStaticText(201,50,100,80,40,"")
0360     #Text2! = #Window!.addStaticText(202,50,130,80,40,"count: ")
0370     #Counter! = #Window!.addStaticText(203,140, 130, 60,40, "0")
0380     #InputE! = #Window!.addInputE(204, 50, 170, 60,40,"")
0390     #Button!.setCallback(#Button!.ON_BUTTON_PUSH,#this!, "toggle")
0400   methodend
0410   method public void show()
0420     #Window!.setCallback(#Window!.ON_CLOSE, "doReturn")
0430     process_events
0440 doReturn:
0450     #Window!.destroy()
0460     methodRet
0470   methodend
0480   method public void toggle(BBjButtonPushEvent event!)
0490     declare BBjString A$
0500     A$ = "no"
0510     if #Active then
0520        A$ = "yes"
0530        #Active = 0
0540     else
0550        #Active = 1
0560     endif
0570     #Text!.setText(A$)
0580     #ClickCount = #ClickCount + 1
0590     #Counter!.setText(str(#ClickCount))
0600   methodend
0610   method public void registerEvents(Reporter reporter!)
0620     #InputE!.setCallback(#Button!.ON_GAINED_FOCUS,reporter!,"report")
0630     #InputE!.setCallback(#Button!.ON_LOST_FOCUS,reporter!,"report")
0640   methodend
0650 classend
0660 class public Reporter
0670   method public void report(BBjGainedFocusEvent p_event!)
0680     print "focus gained on: ", p_event!.getControl()
0690   methodend
0700   method public void report(BBjSysGuiEvent p_event!)
0710     print "unknown event ", p_event!," reported on ", p_event!.getControl()
0720   methodend
0730 classend
```

Some things to notice

- A program can register a callback for an event where the callback is a method on a custom object (as opposed to a label within a program). In this sample we register a callback for ON_BUTTON_PUSH that causes the method **toggle()** to be called on the custom object **this!**. We also register ON_LOST_FOCUS and ON_GAINED_FOCUS callbacks with the methods of a different custom object.
- When registering a custom object method for a callback, the method must accept a single parameter and the event for which the method is being registered must be assignable to the type of that parameter.

- We can register the method **toggle(BBjButtonPushEvent p_event!)** for the ON_BUTTON_PUSH event because the type of parameter is the same as the type of the event that is generated.
- We can register the method **report(BBjSysGuiEvent p_event!)** for the ON_LOST_FOCUS event because the event that is generated (a BBjLostFocusEvent) extends the parameter type of BBjSysGuiEvent.
- We cannot register the method **toggle(BBjButtonPushEvent p_event!)** as the callback for ON_LOST_FOCUS event because the event that is generated (a BBjLostFocusEvent) can not be assigned to BBjButtonPushEvent.
- Within a method we can register a label as the callback (as we do in **show()**) but the label must be within the method where setCallback is being called. Code within a method can not find labels that are not within the same method.
- The callback for ON_WINDOW_CLOSE destroys the window and then uses **methodret** to return to the calling program. After returning from **show()**, we are no longer doing **process_events**.
- We are able to gather information while displaying the dialog and then retrieve that information from the dialog after the window has been destroyed.
- A **declare** statement can be used within a method body (as in **Dialog.toggle()**) to define the type of a local variable within that method body.

Defining Interfaces: using Interface, interfaceEnd, and implements

In this section – We define two interfaces **Weighable** and **Edible** and a number of custom classes, all of which implement the **Weighable** interface and some of which implement **Edible**. We create a **Scale** class that has a single method **weigh()** that accepts a **Weighable** and a class **Nutritionist** that has one method for measuring **Edibles** and a different method for measuring objects that are not **Edibles**.

```
0010 rem weight.src
0020 declare Elephant Elephant!
0030 declare Mouse Mouse!
0040 declare Fish Fish!
0050 declare Butterfly Butterfly!
0060 declare Carror Carot!
0070 declare Watermelon Watermelon!
0080 declare Rock Rock!
0090 declare Salt Salt!
0100 declare BBjNumber total
0110 Elephant! = new Elephant()
0120 Mouse! = new Mouse()
0130 Fish! = new Fish()
0140 Butterfly! = new Butterfly()
0150 Carrot! = new Carrot()
0160 Watermelon! = new Watermelon()
0170 Rock! = new Rock()
0180 Bicycle! = new Bicycle()
0190 Salt!= new Salt()
0200 total = 0
0210 total = total + Scale.Weigh(Elephant!)
0220 total = total + Scale.Weigh(Mouse!)
0230 total = total + Scale.Weigh(Fish!)
0240 total = total + Scale.Weigh(Butterfly!)
```

```
0250 total = total + Scale.Weigh(Carrot!)
0260 total = total + Scale.Weigh(Watermelon!)
0270 total = total + Scale.Weigh(Rock!)
0280 total = total + Scale.Weigh(Bicycle!)
0290 print
0300 print "Scale has weighed a total of:", total
0310 print
0320 print
0330 input "Press [ENTER] to see what the nutritionist says:",*
0340 print
0350 Nutritionist.Measure(Elephant!)
0360 Nutritionist.Measure(Mouse!)
0370 Nutritionist.Measure(Fish!)
0380 Nutritionist.Measure(Butterfly!)
0390 Nutritionist.Measure(Carrot!)
0400 Nutritionist.Measure(Watermelon!)
0410 Nutritionist.Measure(Rock!)
0420 Nutritionist.Measure(Bicycle!)
0430 Nutritionist.Measure(Salt!)
0440 interface public Weighable
0450    method public BBjNumber getWeight()
0460    method public BBjString Name()
0470 interfaceend
0480 class public Scale
0490   method public static BBjNumber Weigh(Weighable weighable!)
0500     weight = weighable!.getWeight()
0510     print weighable!.Name(), " weighs", weight
0520     methodret weight
0530   methodend
0540 classend
0550 interface public Edible
0560   method public BBjNumber getCalories()
0570   method public BBjString Name()
0580 interfaceend
0590 class public Nutritionist
0600   method public static BBjNumber Measure(Edible food!)
0610     print food!.Name(), " contains ", food!.getCalories(), " calories"
0620     methodret food!.getCalories()
0630   methodend
0640   method public static BBjNumber Measure(java.lang.Object notFood!)
0650     print notFood!, " is not edible"
0660       methodret 0
0670     methodend
0680 classend
0690 class public Animal implements Weighable
0700   field private BBjNumber Mammal
0710   field private BBjNumber Weight
0720   field private  BBjString Name$
0730   method protected Animal(BBjString name$, BBjNumber isMammal, BBjNumber weight)
0740     #Name$ = name$
0750     #Mammal = isMammal
0760     #Weight = weight
0770   methodend
0780   method public BBjNumber getWeight()
0790     methodret #Weight
0800   methodend
0810   method public BBjString Name()
0820     methodret #Name$
0830   methodend
0840 classend
0850 class public Flora implements Weighable
0860   field private BBjNumber IsFruit
0870   field private BBjNumber Weight
0880   field private  BBjString Name$
0890   method protected Flora(BBjString name$, BBjNumber isFruit, BBjNumber weight)
0900     #Name$ = name$
0910     #IsFruit = isFruit
0920     #Weight = weight
0930   methodend
0940   method public BBjNumber getWeight()
0950     methodret #Weight
```

```
0960   methodend
0970   method public BBjString Name()
0980     methodret #Name$
0990   methodend
1000 classend
1010 class public Elephant extends Animal implements Weighable
1020   method public Elephant()
1030     #super!("Bimbo", 1, 15000)
1040   methodend
1050 classend
1060  class public Mouse extends Animal
1070    method public Mouse()
1080      #super!("Micky", 1, .025)
1090    methodend
1100 classend
1110 class public Fish extends Animal implements Edible
1120   method public Fish()
1130     #super!("Nemo", 0, 2.3)
1140   methodend
1150   method public BBjNumber getCalories()
1160     methodret 185
1170   methodend
1180 classend
1190 class public Butterfly extends Animal
1200   method public Butterfly()
1210     #super!("butterfly", 0, .00004)
1220   methodend
1230 classend
1240 class public Carrot extends Flora implements Edible
1250   method public Carrot()
1260     #super!("carrot", 0, .2)
1270   methodend
1280   method public BBjNumber getCalories()
1290     methodret 13.5
1300   methodend
1310 classend
1320 class public Watermelon extends Flora implements Edible
1330   method public Watermelon()
1340     #super!("watermelon", 1, 3.1)
1350   methodend
1360   method public BBjNumber getCalories()
1370     methodret 137
1380   methodend
1390 classend
1400 class public Rock implements Weighable
1410   method public BBjNumber getWeight()
1420      methodret  5.7
1430   methodend
1440   method public BBjString Name()
1450      methodret "Rock"
1460   methodend
1470 classend
1480 class public Bicycle implements Weighable
1490   method public BBjNumber getWeight()
1500     methodret  23.87
1510   methodend
1520   method public BBjString Name()
1530     methodret "Bicycle"
1540   methodend
1550 classend
1560 class public Bicycle implements Weighable
1570   method public BBjNumber getWeight()
1580      methodret  23.87
1590   methodend
1600 classend
1610 class public Salt implements Edible
1620   method public BBjNumber getCalories()
1630     methodret .0002
1640   methodend
1650   method public BBjString Name()
1660     methodret "Salt"
```

```
1670   methodend
1680 classend
```

Some things to notice

- The **Scale** class does not know about **Animal** or **Butterfly**, **Carrot,** or **Rock**. The only thing that **Scale** understands is **Weighable**. **Scale** can weigh anything that implements **Weighable**.
- If a class (like **Animal**) implements an interface, then all its subclasses also implement that interface.
- A class can implement more than one interface.
- When **Nutritionist** measures an **Edible** it will tell the calories of the **Edible**. When **Nutritionist** measures any other object it reports that it is not **Edible**.

Constructor: Order of Execution of Constructors, superConstructor, Initializers and Static Initializers

In this section – We explore constructors and field initialization. There are number of things the developer needs to understand about constructors and initializers and the order in which they are executed.

If the custom class does not explicitly define a constructor, BBj will define a default no-arg constructor.

If a custom class **A** has a superclass **B**, every constructor of **A** will call a constructor of **B**. If a constructor of **A** does not explicitly call a constructor of **B**, then BBj will implicitly call the default no-arg constructor of **B**. The constructor of **B** is executed before the constructor of **A**.

All field initializers of a custom class are executed before the constructor code is executed.

All static field initializers of a custom class are executed before any non-static field initializers for that custom class.

Static field initializers of a given custom class are only executed once in any BBj session.

```
0010 class public Vehicle
0020   field public BBjString Color$ = "BLACK"
0030   field protected BBjNumber MaxSpeed = 18
0040   field public BBjString Description$ = str(#SerialNumber) + " Color: " + #Color$ + "
MaxSpeed: " + str(#MaxSpeed)
0050   field protected static BBjNumber SerialNumber = 0
0060  method public BBjNumber getMaxSpeed()
0070    methodret #MaxSpeed
0080  methodend
0090 classend
0100 declare Vehicle transport!
0110 transport! = new Vehicle()
0120 print transport!.getDescription()
0130 class public OldCar extends Vehicle
0140   field public BBjNumber MilesPerGallon = 15
0150   method private OldCar()
0160   methodend
0170   method public OldCar(BBjNumber maxSpeed)
```

```
0180     print "in constructor " + #getDescription()
0190     #setSerialNumber(#getSerialNumber()+1)
0200     #setMaxSpeed(maxSpeed)
0210   methodend
0220 classend
0230 setErr aLabel
0240 declare OldCar aCar!
0250 aCar! = new OldCar()
0260 escape
0270 aLabel:
0280 print " constructor OldCar() not visible"
0290 seterr 0
0300 aCar! = new OldCar(45)
0310 print aCar!.getDescription()
0320 class public NewCar extends OldCar
0330   method private NewCar()
0340   methodend
0350   method public NewCar(BBjNumber maxSpeed)
0360     #super!(maxSpeed)
0370   methodend
0380   method public NewCar(BBjNumber maxSpeed, BBjNumber milesPerGallon)
0390     #this!(maxSpeed)
0400     print "legacy description: ", #getDescription()
0410     #setMilesPerGallon(milesPerGallon)
0420     #setDescription(#newDescription())
0430     print "new description: ", #getDescription()
0440   methodend
0450   method protected BBjString newDescription()
0460     a$ = str(#getSerialNumber()) + " shiny " + #getColor() + " and gets " +
str(#getMilesPerGallon()) + " miles per gallon"
0470     methodret a$
0480   methodend
0490 classend
0500 declare NewCar bCar!
0510 bCar! = new NewCar(95, 48)
0520 print bCar!.getDescription()
0530 class public RaceCar extends NewCar
0540 method public RaceCar(BBjNumber maxSpeed, BBjNumber milesPerGallon)
0550   #super!(maxSpeed)
0560   #setMilesPerGallon(milesPerGallon)
0570   #setColor("RED")
0580   #setDescription(#newDescription())
0590   methodend
0600 classend
0610 declare RaceCar carC!
0620 carC! = new RaceCar(217, 3.2)
0630 print carC!.getDescription()
```

Some things to notice:

- In the statement **`transport! = new Vehicle()`**, we call the default no-arg constructor. Because this constructor is not defined in the program it was generated by BBj. A custom class always has a no-arg constructor.
- The field initializers are executed in the order in which they appear. So when **Vehicle.Description$** is initializing, it can use the already initialized values of the fields **Vehicle.Color$** and **Vehicle.MaxSpeed**.
- Because **Vehicle.SerialNumber** is a static field, it initializes before **Vehicle.Description$** even though it appears after the **Vehicle.Description$** in the code. So the initializer for **VehicleDescription$** is able to access the already initialized value of **Vehicle.SerialNumber**.
- The default no-arg constructor for OldCar has been overridden and declared private, so the statement **`aCar! = new OldCar()`** results in an error.

- The constructor **`OldCar(BBjNumber maxSpeed)`** sets the values of **#SerialNumber** and **#MaxSpeed**. Since the constructor is executed after the field initializers, the values that are set in the constructor are reflected in later calls to **OldCar.getDescription()**.
- The constructor **`NewCar(BBjNumber maxSpeed)`** can explicitly call a specific super constructor using the syntax **`#super!(maxSpeed)`**.
- The constructor **`NewCar(BBjNumber maxSpeed, BBjNumber milesPerGallon)`** calls a different constructor of **NewCar** using the syntax **`#this!(maxSpeed)`**.
- A constructor can call methods of the class being constructed. So, for example, the constructor **`NewCar(BBjNumber maxSpeed, BBjNumber milesPerGallon)`** contains the statement **`#Description$ = #newDescription()`**. When the constructor is complete, the value of **Description$** will contain the return value of **newDescription()**.

Conclusion

Custom objects bring the power of object-oriented programming to the BBx language. This introduction should contain enough information so that the BBj developer can begin creating and using custom objects within BBj code.

Post additional questions to bbj-developer@basis.com. To join this discussion forum, subscribe online.