Object-Oriented Programming

With BBj Custom Classes, Java Classes, Event Objects, and BBjAPI()

Brian Hipple, Quality Assurance Supervisor

BASIS International Ltd.

BASIS International

Business Runs on BASIS®

Getting Started

- Logistics
- Teacher profile
- Expectations

Business Runs on BASIS®

Logistics

- Hours
- Breaks
- Food
- Facilities
- Materials
- Handouts

Business Runs on BASIS®

Teacher Profile

- Brian Hipple
  - Brings 17 years programming experience
  - BBx, Java, Visual C++, C
  - Developed BBj/BBx product lines
  - BASIS Test Engineering Supervisor
  - Worked for several Fortune 500 companies
  - Java Web Services certified

Class Outline

- ▸ From earlier versions of BBj
  - Object variables (X!)
  - Interacting with Java
  - BBjAPI() and BBj GUI Controls
- ▸ New for BBj 6.0
  - BBj Custom Objects
  - Type Checking
  - BBj Event Objects
- ▸ The focus will be on BBj Custom Objects

Business Runs on BASIS®

Module: Using Objects in BBj

Review of BBj Object-Based Features

Business Runs on BASIS®

Object Variables

- ▸ BBx variable types include
  - X$ is a string variable
  - X is a numeric variable
  - X% is an integer variable
- ▸ BBj also supports object variables
  - X! is an object variable
- ▸ Object variables are universal holders
  - X! = "Hello, world"
  - X! = 12.95
  - X! = I%

Business Runs on BASIS®

Object Variables as Strings

- ▸ Assign a string to an object variable
  - X!="The quick brown fox"
- ▸ These are ok:
  - PRINT X!
  - PRINT LEN(X!)
- ▸ But this is an error (why?)
  - DIM X!:"STATE:C(2*),NAME:C(50*)"

Business Runs on BASIS®

Object Variables as Numbers

- Assign a number to an object variable
  - X!=12.95
- These are ok:
  - PRINT X!:"$###.00"
  - A=X!+Y+Z!
  - A=MAX(X!,Y)
- But this is an error (why?)
  - INPUT X!

Business Runs on BASIS®

Object Variables and Java

- Object variables are used with Java classes
  - Now! = new java.util.Date()
  - Map! = new java.util.HashMap()
  - List! = new java.util.ArrayList()
- See the Java Collections classes for details

  http://java.sun.com/j2se/1.5.0/docs/guide/collections/
- But don't use Java GUI ("Swing") classes!
  - BBj uses client/server technology to remote GUI calls
  - Calling Java Swing classes bypasses BBj client/server

Business Runs on BASIS®

Object Variables and BBjAPI()

- BBjAPI() is the entry point for creating and manipulating BBj objects and object-based features
  - Object-oriented interface for GUI controls
  - Event callback syntax

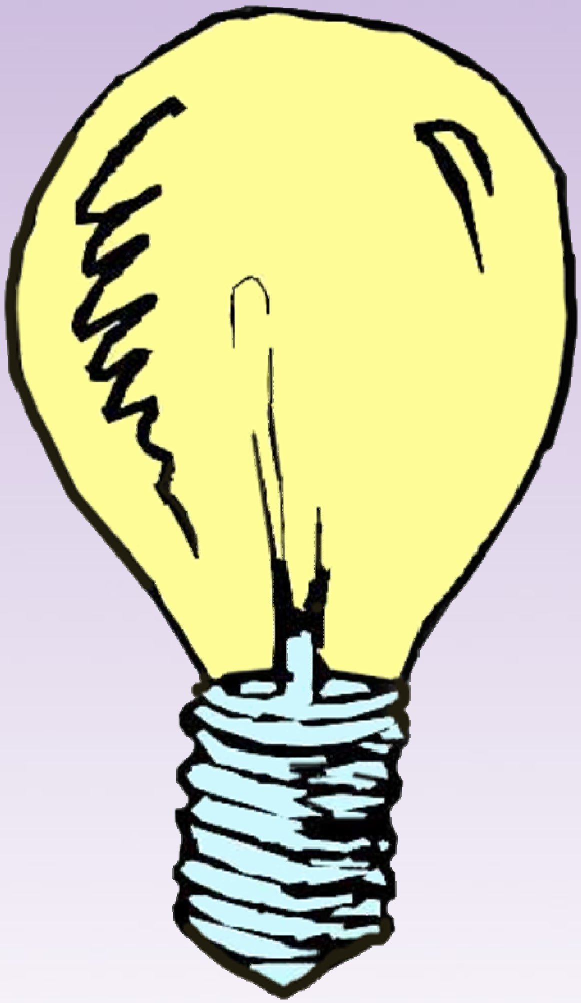
```
sysgui! = bbjapi().getSysGui()
vector! = bbjapi().makeVector()
fs! = bbjapi().getFileSystem()
bbjapi().createTimer(id,seconds,label)
```

- Uses factory type interface (create, get, make) instead of new

Business Runs on BASIS®

What Is An Object?

- Objects model real-world "things"
- A light bulb object has
  - An internal state (on/off)
  - Some way for the outside world to change that state



Business Runs on BASIS®

What Is An Object?

- More formally, an object is:
  - A self-contained program module that carries around both internal data and the ability to act on that data
- All objects:
  - Have a type (defined by their **class**)
  - Have internal state (**fields**)
  - Interact through controlled interfaces (**methods**)

Business Runs on BASIS®

Why Use Objects?

- ▸ Objects closely model the real world
  - Application design is translated to programming code in a more direct way than with older programming models
- ▸ Self-contained
  - Each object carries both methods and internal state
  - Building blocks for constructing larger applications
- ▸ Controlled access
  - Interaction is through defined methods
  - Internal data is hidden (no variable conflicts!)
  - Internal implementation can be evolved over time

Business Runs on BASIS®

Classes Versus Objects

- Classes and Objects are not the same
- List! = **new** java.util.ArrayList()
  - List! is an object
  - java.util.ArrayList is a class
- The **new** operator creates a new object
- A class is a blueprint for building an object
- An object is a specific *instance* of a class

Business Runs on BASIS®

Classes Versus Objects

- Same class, different objects
- For example, a HashMap is a Java data structure for manipulating key/value pairs
- The following objects are different instances of the HashMap class:
  - TaxTable! = new java.util.HashMap()
  - Country! = new java.util.HashMap()
  - Inventory! = new java.util.HashMap()

Business Runs on BASIS®

Object Syntax Notes

- Objects are created with the 'new' operator
  - List! = **new** java.util.ArrayList()
- Methods are accessed with the '.' operator
  - List!**.**add("xyzzy")
- Method names are case sensitive!
  - List!.Add("xyzzy"); rem ' Error!

Business Runs on BASIS®

Object Variables: Ref. vs. Value

- Object variables are references (pointers)
  - String and numeric variables are copied "by value"
    - X$=Y$; REM ' copies the value of Y$ to X$
    - X=Y; REM ' copies the value of Y to X
  - But object variables are copied "by reference"
    - List! = new java.util.ArrayList()
    - Copy! = List!
    - List!.add("xyzzy")
    - PRINT Copy!.get(0); REM ' returns "xyzzy"
  - Some classes support the clone() method to make a copy

Business Runs on BASIS®

Module: BBj Custom Classes

Defining and Using Classes in BBj

BASIS International

Business Runs on BASIS®

BBj Custom Classes

- BBj has always provided an object-oriented syntax; what is new in BBj 6.0?
- In earlier versions of BBj, you could:
  - Directly access the large library of standard Java classes
  - Directly access many standard classes provided by BASIS through BBjAPI()
  - Write GUI applications using a simple object-oriented syntax
  - Interact with your own custom classes written in Java
- In BBj 6.0, you can create classes directly in BBj itself - BBj Custom Classes

Business Runs on BASIS®

A Simple BBj Class

- A BBj custom class is a block of code bracketed by the keywords **class** and **classend**:

```
class public Sample
classend
```

- A class is either **public** (visible outside the current program) or **private** (not visible)
- The class name is **case sensitive**

A Simple BBj Class

- To define a BBj class:

```
class public Sample
classend
```

- To create a new object of that class:

```
mySample! = new Sample()
```

Business Runs on BASIS®

External Classes

- Classes can be defined in another file
- Refer to class as ::filename::Classname()
- File is located using standard prefix rules

```
rem ' SampleClass.bbj
class public Sample
classend

rem ' Application.bbj
Sample! = new ::SampleClass.bbj::Sample()
```

The USE Verb

- Tells BBj where to find an external class
- Preferred way to refer to external classes

```
rem ' SampleClass.bbj
class public Sample
classend

rem ' Application.bbj
use ::SampleClass.bbj::Sample
Sample! = new Sample()
```

The USE Verb

- Tells BBj where to find a Java class
- No need to keep typing the full name

```
use java.util.ArrayList
List! = new ArrayList()
```

- The **use** statements are resolved as the program is loaded; they can go anywhere in the program

BASIS International

Business Runs on BASIS®

Exercise 1

- ❑ Create all programs in the directory `/training/BBjObjects/exercises`
- ❑ Write a program named **`exercise1`** that defines a minimal BBj class called Sample
- ❑ Add code outside the class definition to create an instance of this class and print the resulting object
- ❑ Copy only the class definition to **`exercise1a`**
- ❑ Copy everything else to **`exercise1b`**
- ❑ Add a use statement to **`exercise1b`** to import the class definition from **`exercise1a`** and confirm that **`exercise1b`** works the same as **`exercise1`**

Module: BBj Custom Classes

Fields

Business Runs on BASIS®

Fields

- Object state information is stored in fields

```
class public Sample
field public BBjNumber Count = 1
classend

mySample! = new Sample()
```

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

- Definition starts with the **field** keyword
- Must begin on a new line
- Syntax error if outside a class definition

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

- Visibility is one of
  - Public: field is visible to users of the class
  - Private: field is visible only within the class itself
- Data Hiding is a key object-oriented concept

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

- Data type can be any BBj or Java class
- Special predefined types include:
  - BBjNumber, BBjString and BBjInt (corresponding to the traditional BBj variable types: N, S$, I%)
  - Built-in BBjAPI objects, including BBjWindow, BBjButton, etc.

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

- Field name is **Count**
- Follows BBx/BBj variable naming rules
- Field names are case sensitive

Business Runs on BASIS®

Fields

- What does this mean?

```
field public BBjNumber Count = 1
```

- An optional initial value can be specified
- If omitted, initial value will be:
  - BBjString = ""
  - BBjNumber and BBjInt = 0
  - All other objects = **null()**

Business Runs on BASIS®

Fields

- Field type is specified as a class name (which might be a special built-in type)
- Normal BBj variable naming rules apply

```
field public BBjNumber Number
field private BBjInt Counter%
field public BBjString Message$
field public BBjWindow Window!
field private Sample Sample!
field public ArrayList List!
```

Exercise 2a

- ❑ Write a program `exercise2` that defines a class called Address
- ❑ This class should include string fields for Name, Street and City and a numeric field for the Zip code
- ❑ Add code outside the class to create a new Address object

Module: BBj Custom Classes

Field Accessors

Business Runs on BASIS®

Field Accessors

- How do you manipulate the **Count** field?

```
class public Sample
field public BBjNumber Count = 1
classend

mySample! = new Sample()
```

Field Accessors

- Access to fields is strictly controlled
- **Accessors** are generated with the names **set*FieldName*()** and **get*FieldName*()**

```
mySample! = new Sample()
mySample!.setCount(23)
print mySample!.getCount()
```

Field Accessors

- Field visibility is either **public** or **private**

```
class public Sample
field public BBjNumber Count
field private BBjNumber X
classend
mySample! = new Sample()
print mySample!.getCount(); rem ok
print mySample!.getX(); rem Error!
```

Business Runs on BASIS®

Field Accessors

- BBj programs have no problem with these two variables existing in the same program:

```
Rate = 42
Rate$ = "xyzzy"
```

- But this is illegal:

```
field public BBjNumber Rate = 42
field public BBjString Rate$ = "xyzzy"
```

- Why?

Business Runs on BASIS®

Field Accessors

- Field Accessors demonstrate these key object-oriented concepts:
  - Access control
    - The developer can choose which fields will be accessible from outside the class
  - Data hiding
    - The implementation can be evolved more easily if implementation-specific fields are invisible outside the class

Business Runs on BASIS®

Exercise 2b

- ❑ Edit your `exercise2` program
- ❑ Add code outside the class to set all fields of your Address object using the automatically generated field accessors

Module: BBj Custom Classes

Methods

Business Runs on BASIS®

Methods

- Objects interact with the outside world through **methods**

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Methods

- A method is a block of code within a class
- It starts with the keyword **method**
- It ends with the keyword **methodend**

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Methods

- The visibility (**public** or **private**) indicates whether the method will be accessible from outside the class

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Methods

- The return type specifies what kind of value the method returns (if any)
- The keyword **void** indicates that no value is returned

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Methods

- The method name follows normal variable naming rules
- The method name is case sensitive

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Business Runs on BASIS®

Methods

- Parameters (if any) are listed in parentheses after the name
- This sample method takes no parameters

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend
```

Business Runs on BASIS®

Invoking a Method

```
class public Sample
  method public void Hello()
    print "Hello, world!"
  methodend
classend

Sample! = new Sample()
Sample!.Hello()
```

methodret

- **methodret** returns a value from a method
- The value must be of the defined type

```
class public Sample
  method public BBjString Hello()
    methodret "Hello, world!"
  methodend
classend
Sample! = new Sample()
Hello$ = Sample!.Hello()
```

Method Parameters

- A method can take parameters
- Each parameter has a type and a name

```
class public Sample
  method public void say(BBjString msg$)
    print msg$
  methodend
classend
Sample! = new Sample()
Sample!.say("Hello world!")
```

Method Parameters

- Methods can take multiple parameters

```
class public Sample
  method public void say(BBjString msg$,
                         BBjNumber num)
    print msg$,num
  methodend
classend
Sample! = new Sample()
Sample!.say("Hello world!",23)
```

Business Runs on BASIS®

Field Accessors Revisited

- Custom accessors can limit field access
- In this example, the user can call getCount() but not setCount() (Why?)

```
class public Sample
  field private BBjNumber Count = 1
  method public BBjNumber getCount()
    methodret #Count
  methodend
classend
```

Business Runs on BASIS®

Referring to Fields

- Inside a class, fields can be referenced with the syntax **#Fieldname**
- This distinguishes them from other variables

```
class public Sample
  field private BBjNumber Count = 1
  method public BBjNumber getCount()
    methodret #Count
  methodend
classend
```

Exercise 2c

- ❑ Edit your `exercise2` program.
- ❑ Change one of the fields in your Address class from public to private.
- ❑ Add the necessary accessor methods to enable you to continue to get and set this private field.
- ❑ Add a print() method your Address class. This method should print all field values.
- ❑ Add code outside the class to use the print() method to print the address.
- ❑ Test this final version of your Address class.

Module: BBj Custom Classes

Overloaded Methods

Business Runs on BASIS®

Overloaded Methods

```
class public Sample
  method public void say(BBjString msg$)
    print msg$
  methodend
  method public void say(BBjString msg$,
                         BBjNumber count)
    for i=1 to count
      print msg$
    next i
  methodend
classend
```

Business Runs on BASIS®

Overloaded Methods

- Multiple methods with the same name are said to be *overloaded*
- Each of the overloaded methods has a unique *signature* derived from the method name and the number and types of parameters
- BBj picks the correct version by looking for a match based on the signature
- The return type is *not* part of the signature (Why?)

BASIS International

Business Runs on BASIS®

Overloaded Methods

- This set of overloaded methods is valid

```
method public void say(BBjString say$,BBjString more$)
method public void say(BBjString say$,BBjNumber times)
method public void say()
method public void say(BBjNumber words)
```

- This set is not valid (Why?)

```
method public void say(BBjString say$)
method public BBjString say(BBjString say$)
```

Overloaded Methods

- What is wrong with this program?

```
class public Sample
  field public BBjNumber Count
  method public BBjString getCount()
    methodret str(#Count)
  methodend
classend
Sample!=new Sample()
print Sample!.getCount()
```

Overloaded Methods

- **BBjString getCount()** conflicts with an automatically generated accessor method

```
class public Sample
  field public BBjNumber Count
  method public void setCount(BBjNumber Count)
    #Count = Count
  methodend
  method public BBjNumber getCount()
    methodret #Count
  methodend
  method public BBjString getCount()
    methodret str(#Count)
  methodend
classend
Sample!=new Sample()
print Sample!.getCount()
```

Module: BBj Custom Classes

Constructors

Business Runs on BASIS®

Constructors

- Constructors are special methods that are implicitly called when a class is created
- BBj generates a default constructor like:

```
class public Sample
    method public Sample()
    methodend
classend
```

Constructors

- Constructor takes the name of the class
- Constructor does not show a return type

```
class public Sample
   method public Sample()
   methodend
classend
```

Business Runs on BASIS®

Constructors

- Constructors can take parameters

```
class public Sample
   field public BBjNumber Num
   method public Sample(BBjNumber Num)
      #Num = Num
   methodend
classend
Sample! = new Sample(42)
print Sample!.getNum()
```

Constructors

- A class can have multiple constructors

```
class public Sample
    field public BBjNumber MyNumber
    field public BBjString MyString$
    method public Sample(BBjNumber num)
        #MyNumber = num
        #MyString$ = ""
    methodend
    method public Sample(BBjNumber num, BBjString str$)
        #MyNumber = num
        #MyString$ = str$
    methodend
classend
```

Constructors

- Constructors can call other constructors

```
class public Sample
  field public BBjNumber MyNumber
  field public BBjString MyString$
  method public Sample()
    #this!(0,"")
  methodend
  method public Sample(BBjNumber n)
    #this!(n,"")
  methodend
  method public Sample(BBjNumber n, BBjString s$)
    #MyNumber = n
    #MyString$ = s$
  methodend
classend
```

#this!

- The current instance
  - #this!
- A different constructor
  - #this!(...)
- Fields
  - #MyField = Value
- Methods
  - #doSomething()
  - #this!.doSomething()

Business Runs on BASIS®

Constructors

- If a constructor calls another constructor, it must do so before doing anything else

```
method public Sample()
    rem ' no code can appear here
    #this!(0,"")
methodend
```

Constructors

- The default constructor can be hidden by overriding it with a **private** constructor

```
class public Sample
  field public BBjNumber Num
  method private Sample()
  methodend
  method public Sample(BBjNumber Num)
    #Num = Num
  methodend
classend
```

- Why might you do that?

Module: BBj Custom Classes

Static Fields and Methods

Business Runs on BASIS®

Static Fields

- A class can include **static** fields

```
field private static BBjNumber FileChannel
```

- Shared across all instances of a class within the current BBj session
- Created the first time the class is referenced
- Persist for the lifetime of the BBj session

BASIS International

Business Runs on BASIS®

Static Fields

```
class public Customer
  field private static BBjNumber Channel
  field private BBjNumber CustNum
  field private BBjString Name$
  method public Customer(BBjNumber num,
                         BBjString name$)
    if #Channel = 0 then
       #Channel = unt
       open (#Channel)"cust"
    fi
    #CustNum = num
    #Name$ = name$
  methodend
classend
```

Static Methods

- A class can include static methods

```
method public static BBjNumber getChannel()
   methodret #Channel
methodend
```

Static Methods

- Can be called on the class

```
class public System
  field private static BBjNumber SysGui
  method public Static BBjNumber getSysGui()
    if #SysGui = 0 then
       #SysGui = unt
       open (#SysGui)"X0"
    fi
    methodret #SysGui
  methodend
classend

sysgui = System.getSysGui()
```

Static Methods

- Cannot access non-static fields or methods

```
class public System
  field private BBjString Company$
  method public Static BBjString getCompany()
    methodret #Company$
  methodend
classend

company$ = System.getCompany(); rem ' Error!
```

Module: BBj Custom Classes

Extending Classes

Business Runs on BASIS®

Extending Classes

- A class can **extend** an existing class
- The new class inherits all non-private fields and methods from the class it extends
- The original class is called the superclass
- The new class is called a subclass or derived class

Business Runs on BASIS®

Extending Classes

- This is a base class:

```
class public Base
  field public BBjString MyString$
  method public Base(BBjString str$)
    #MyString$ = str$
  methodend
classend
```

Extending Classes

- This class is derived, or *extended*, from Base

```
class public Derived extends Base
  field public BBjNumber MyNumber
  method public Derived(BBjString s$, BBjNumber n)
    #super!(s$)
    #MyNumber = n
  methodend
  method public void print()
    print #super!.getMyString()
    print #MyNumber
  methodend
classend
```

Business Runs on BASIS®

#super!

- #super! is a reference to the superclass
- Constructor in the superclass
  - #super!(...)
- Fields of the superclass
  - #super!.getFieldName()
  - #super!.setFieldName()
- Methods of the superclass
  - #super!.print()

Business Runs on BASIS®

Extending Classes

- A constructor in a derived class can call a constructor in its superclass
- If a derived class constructor does not explicitly call a superclass constructor, it implicitly calls the default superclass constructor, equivalent to: #super!()
- The call to #super!(), if used, must be the first line in the derived class constructor

protected

- There are actually three visibility levels: **public**, **private** and **protected**

```
field protected BBjNumber Count
method protected BBjNumber getCount()
```

- Protected fields and methods are only accessible from the current class and any subclasses

Business Runs on BASIS®

Visibility Levels

- Public
  - accessible from within class
  - accessible from subclasses
  - accessible from outside class
- Protected
  - accessible from within class
  - accessible from subclasses
- Private
  - accessible only within class

Module: BBj Custom Classes

Interfaces

Business Runs on BASIS®

Interfaces

- An *interface* is a set of methods

```
interface public Printable
   method public BBjString getFormat()
   method public void print()
   ....
interfaceend
```

Interfaces

- A class *implements* an interface
- *Implements* is a promise made by a class that it will implement a set of methods

```
class public CustomerAddress extends Address
  implements Printable
```

  - The class must implement all methods of an interface
  - A class can implement multiple interfaces

```
class public CustomerAddress extends Address
  implements Printable,Editable,Searchable
```

Business Runs on BASIS®

Interfaces

- An interface acts much like a class

```
method public void printLabel(Printable p!)
    print p!.getFormat()
    p!.print()
    .....
```

- Interfaces only have methods, not fields
- The printLabel(Printable) method accepts any object that implements the Printable interface

Interface Sample

```
class public CustomerAddress Extends Address implements
   Printable
   field private BBjNumber CustNum
       method public CustomerAddress(BBjString p_name$)
             #super!(p_name$)
       #CustNum = #CustNum + 1
   methodend
   method public BBjString getFormat()
             metodret “pdf”
   methodend
   method public void print()
             print "CustNum: ",#CustNum
       #super!.print()
   methodend
classend
```

Business Runs on BASIS®

Module: BBj Custom Classes

Error Handling

Business Runs on BASIS®

Plan For The Unexpected

- Successful classes will be reused by many different programs in the future
- Some of those programs might pass data values into the class that weren't expected
- The class should respond reasonably, even in the face of unanticipated inputs
- Data passed into a class should be sanity checked before being accepted

Business Runs on BASIS®

Error Handling

- ERR= can be used to trap errors
- This version will fail if given bad data:

```
method public void makeNum(BBjString value$)
   methodret num(value$)
methodend
```

- This version recovers from bad data:

```
method public void makeNum(BBjString value$)
   methodret num(value$,err=*next)
   methodret 0
methodend
```

Business Runs on BASIS®

Error Handling

- SETERR can be used to trap errors

```
method public void makeNum(BBjString value$)
    seterr bad_data
    methodret num(value$)
bad_data:
    methodret 0
methodend
```

- The label must exist inside the method

Business Runs on BASIS®

Error Handling

- The **THROW** verb can pass a chosen error to the caller

```
class public Convert
   method public static BBjNumber toNum(BBjString str$)
      methodret num(str$,err=*next)
      throw "Not numeric: '"+str$+"'",41
   methodend
classend

print Convert.toNum("x")
```

THROW

- Throw errors for exception conditions

```
method public void drive(BBjNumber p_miles)
   if #GasAmount = 0 then
      throw "Cannot drive without gas",41
   fi
   gasUsed = p_miles / #GasMileage
   if gasUsed <= #GasAmount then
      #GasAmount = #GasAmount - gasUsed
      print "Drove",p_miles," miles on",gasUsed, " gallons"
   else
      milesDriven = #GasAmount * #GasMileage
      #GasAmount = 0
      throw "Out of gas at "+str(milesDriven)+" miles",40
   fi
methodend
```

THROW

- Constructors should sanity-check data

```
   method public Car(BBjNumber power, BBjNumber capacity,
:                    BBjNumber mileage, BBjString color$)
      if capacity < 10 then
         throw "Gas Tank too small",41
      fi
      if mileage < 1 or mileage > 50 then
        throw "Unbelievable Gas Mileage",41
      fi
    rem ......
   methodend
```

Business Runs on BASIS®

THROW

- Catch and throw an error from a method

```
method public void worker()
    seterr do_error
    rem ' ...do work here...
    Methodret
    do_error:
    rem Pass on the error with more info
    throw "In worker():"+errmes(-1),err
methodend
```

Untrapped Errors In Methods

- Untrapped errors are handled like errors in public programs
- The behaviour depends on the setting of SETOPTS byte 1, bit 08
- When set, the program drops to console mode on the line where the error occured.
- When not set, the error is passed back to the method invoker

Business Runs on BASIS®

Untrapped Errors In Methods

```
seterr errtrap
setopts $00$
   Sample! = new Sample()
   stop
errtrap:
   print errmes(-1),err

class public Sample
   method public Sample()
      a = sqr(-1)
   methodend
classend
```

Module: Type Checking

Business Runs on BASIS®

Type Checking

- Define runtime versus compile-time type checking
- Earlier versions of BBj only check at runtime
- New options allow for compile-time type checking
  - **DECLARE** verb
  - `bbjcpl -t -W`
- Compile-time type checking helps to spot potential errors before they get into the field

BASIS International

Business Runs on BASIS®

Type Checking

- BBj custom class references are type-checked at compile time
- Use the new **declare** verb to enforce type-checking of object variables

```
declare java.util.HashMap Map!
Map! = new java.util.ArrayList(); rem ' Error!
```

- The BASIS IDE refers to declare statements to implement code completion

Business Runs on BASIS®

Type Checking

- Class field references are type checked
- The following code generates an error attempting to assign a BBjTopLevelWindow object to a BBjChildWindow field

```
field private BBjChildWindow MyWin!
sysgui! = bbjapi().getSysGui()
#MyWin! = sysgui!.addWindow(10,10,100,100,"")
```

Type Checking

- Method parameters are type checked
- The following code generates an error attempting to pass a BBjTopLevelWindow object to a BBjChildWindow parameter

```
sysgui! = bbjapi().getSysGui()
win! = sysgui!.addWindow(10,10,100,100,"")
#doWork(win!)
...
method public void doWork(BBjChildWindow win!)
```

Business Runs on BASIS®

Type Checking

- The declare verb enforces type checking
- The following code generates an error attempting to assign a BBjTopLevelWindow object to a BBjChildWindow variable

```
declare BBjChildWindow Win!
sysgui!=bbjapi().getSysGui()
Win! = sysgui!.addWindow(10,10,100,100,"")
```

Type Checking

- Why does this work?

```
field private BBjWindow MyWin!
sysgui! = bbjapi().getSysGui()
#MyWin! = sysgui!.addWindow(10,10,100,100,"")
```

- addWindow() returns a BBjTopLevelWindow
- BBjTopLevelWindow is derived from (extends) BBjWindow
- Base types are compatible with their own derived types

Type Checking

- Why does this **not** work?

```
declare BBjButton myButton!
declare BBjVector v!
open (unt)"X0"
sysgui!=bbjapi().getSysGui()
win! = sysgui!.addWindow(100,100,100,100,"")
v! = bbjapi().makeVector()
v!.addItem(win!.addButton(1,10,10,90,25,"OK"))
myButton! = v!.getItem(0); rem ' error!
```

  - BBjVector::getItem() returns a java.lang.Object
  - A java.lang.Object can't be assigned to a BBjButton variable
  - This is **`Sample15.src`**

BASIS International

Business Runs on BASIS®

Type Checking

- But this works! Why?

```
declare BBjVector v!
open (unt)"X0"
sysgui!=bbjapi().getSysGui()
win! = sysgui!.addWindow(100,100,100,100,"")
v! = bbjapi().makeVector()
v!.addItem(win!.addButton(1,10,10,90,25,"OK"))
myButton! = v!.getItem(0)
```

  - BBjVector::getItem() returns a java.lang.Object
  - A java.lang.Object **can** be assigned to an **undeclared** object variable (myButton!).
  - This is `Sample16.src`

Business Runs on BASIS®

Type Checking

- Objects returned from known methods are type checked

```
Win! = bbjapi().getSysGui().addWindow(...)
doWork(Win!)
method public void doWork(BBjWindow Win!)
    Win!.setTitle("My new title")
methodend
```

- The method call works because BBjTopLevelWindow is a subclass of BBjWindow
- But the setTitle(...) fails because it is not a method of BBjWindow, but only of BBjTopLevelWindow
- This is **`Sample17.src`**

Business Runs on BASIS®

The CAST() Function

- The CAST() function treats an object as an instance of a specific class

```
declare BBjButton myButton!
declare BBjVector v!
v! = new bbjapi().makeVector()
v!.addItem(myWin!.addButton(1,10,10,90,25,"OK"))
myButton! = cast(BBjButton, v!.getItem(0))
```

Module: BBj Custom Classes

Event Objects

Business Runs on BASIS®

Event Objects

- Object-oriented contain event information
- Provide methods to access event information
- Created
  - Automatically when event occurs
  - Programmatically
    - BBjAPI::postCustomEvent, BBjAPI::postPriorityCustomEvent
- Accessed
  - BBjAPI::getLastEvent()
  - As a parameter to object based event handler

BASIS International

Business Runs on BASIS®

Event Objects

- To use a method as an event handler, include an object reference in a callback:

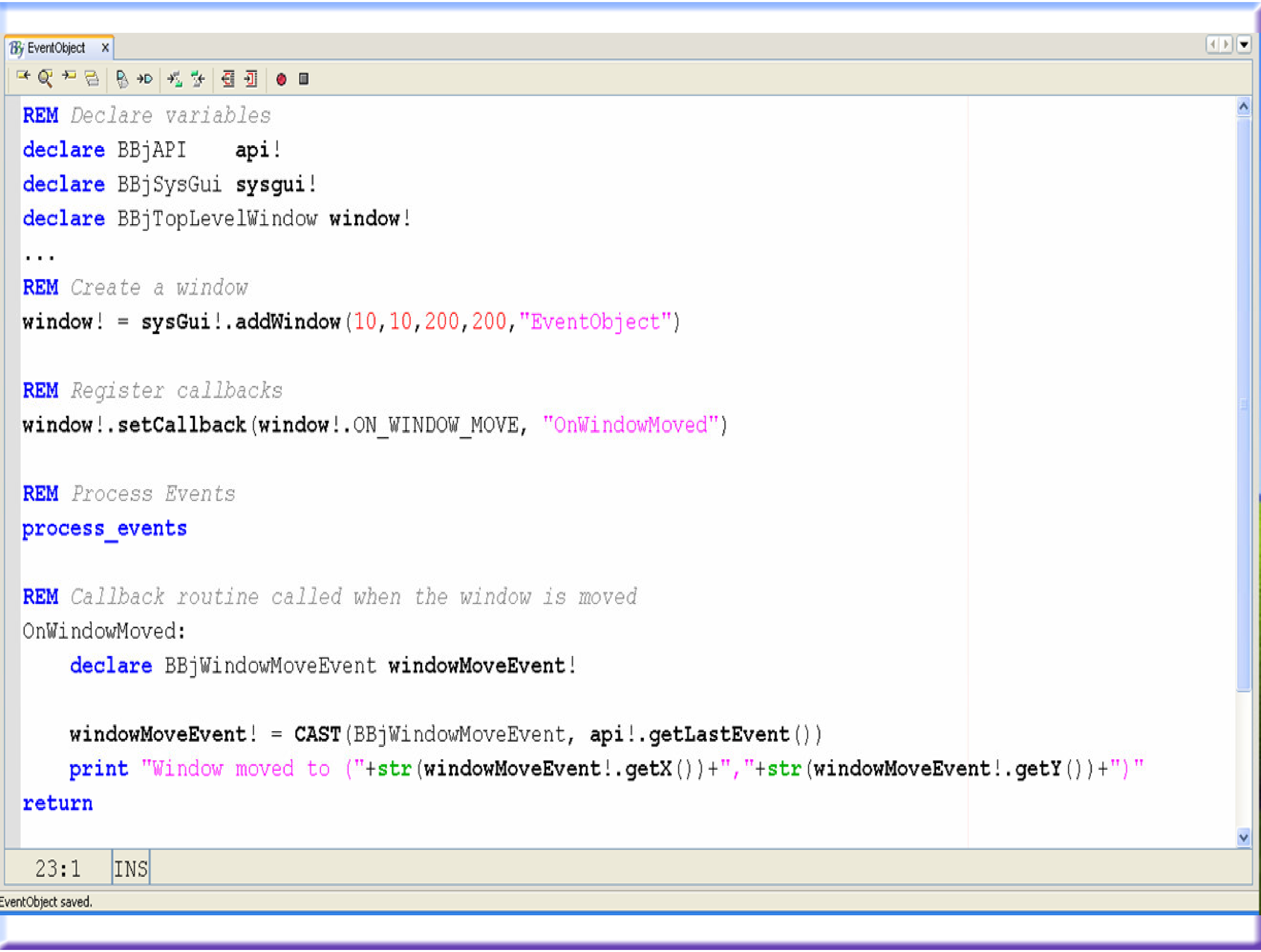
```
#Button!.setCallback(#Button!.ON_BUTTON_PUSH,#this!,"OnOK")
```

- The corresponding event handler is:

```
method public void OnOK(BBjButtonPushEvent event!)
  i=msgbox(event!.toString(),0,"Button Pushed")
methodend
```

- The event handler parameter is a BBjEvent

Business Runs on BASIS®



```
REM Declare variables
declare BBjAPI    api!
declare BBjSysGui sysgui!
declare BBjTopLevelWindow window!
...
REM Create a window
window! = sysGui!.addWindow(10,10,200,200,"EventObject")

REM Register callbacks
window!.setCallback(window!.ON_WINDOW_MOVE, "OnWindowMoved")

REM Process Events
process_events

REM Callback routine called when the window is moved
OnWindowMoved:
    declare BBjWindowMoveEvent windowMoveEvent!

    windowMoveEvent! = CAST(BBjWindowMoveEvent, api!.getLastEvent())
    print "Window moved to ("+str(windowMoveEvent!.getX())+","+str(windowMoveEvent!.getY())+")"
return
```

Event Objects

Refer to the BBj 6.0 documentation for all defined event objects, including a handy conversion table from Visual PRO/5 event strings to the corresponding event objects

Business Runs on BASIS®

Review

- ▸ From earlier versions of BBj
  - Object variables (X!)
  - Interacting with Java
  - BBjAPI() and BBj GUI Controls
- ▸ New for BBj 6.0
  - BBj Custom Objects
  - Type Checking
  - BBj Event Objects

Business Runs on BASIS®

Questions and Answers

Business Runs on BASIS®