

ClientObject Tutorial

May 2008

By

David Wallwork

The purpose of this tutorial is to provide an introduction to the syntax of ClientObjects.

[Introduction](#)

[Syntax of ClientObjects](#)

[Using a ClientObject](#)

[ClientObject vs. Server-side Objects](#)

[Placing a JComponent into a BBjWindow](#)

[Registering a BBj Callback as a Listener on a JComponent](#)

[Mixing ClientObjects and BBjControls to Provide a BBjWindow With a LayoutManager](#)

[Requirements for Class files used to create ClientObjects](#)

[Client Objects and Java Permissions](#)

ClientObject Tutorial

Introduction

[Back to top](#)

Since version 2.0, BBJ® allows programmers to use embedded Java to access the full functionality of the Java language from within a BBJ program. So, for example, a program might use the following code to print the current time.

```
REM print time and TimeZone of server machine

use java.util.Date
use java.text.DateFormat

localDate! = new Date()
localFormat! = DateFormat.getDateTimeInstance()

print localFormat!.format(localDate!), " ",
print localFormat!.getTimeZone().getDisplayName()
```

Code Sample 1: Using embedded (server-side) objects

In many configurations, the user display is on a different machine (the client machine) than that on which the [BBJInterpreterServer](#) is running (the server machine). Because these machines may be in different time zones, this small program may display results that are different from what the programmer intended. This code is executed within the Java Virtual Machine (JVM) of the server and so it will display the time and time zone of the server machine while the programmer may have wanted to display the local time on the client machine.

Prior to BBJ 8.0, there has not been a mechanism by which a program could execute code within the JVM of the client. BBJ 8.0 introduces syntax that provides an RMI (Remote Method Invocation) interface that allows the embedded Java code within a BBJ program to access objects that reside in the client JVM rather than within the server JVM.

In order to access this RMI capability, an @ symbol is appended to a class name to indicate that the class will be accessed in the client JVM rather than in the server JVM.

```
10 REM print time and TimeZone of client machine

20 use java.util.Date
30 use java.text.DateFormat

40 remoteDate! = new Date@()
50 remoteFormat! = DateFormat@.getTimeInstance()

60 print remoteFormat!.format(localDate!), " ",
70 print remoteFormat!.getTimeZone().getDisplayName()
```

Code Sample 2: Using ClientObjects

When this code is executed it will use the time of the client machine to create a new Date Object in the client JVM and will use the locale of the client to create a DateFormat object in the client

JVM. The result of running this program will be to display the time and TimeZone of the client machine rather than the time and TimeZone of the server.

When objects are created in the client JVM those objects are referred to as client-side objects. The server-side variables that reference those client-side objects are referred to as ClientObjects. So, for example, when line 40 in sample 2 is executed a client-side instance of Date is created and the variable remoteDate! is a ClientObject that represents an RMI handle to that client-side object. The ClientObject can be manipulated in the same way as a server-side object.

This paper provides a short tutorial-by-example on how to use ClientObjects. If unfamiliar with the use of Java objects within a BBJ program, read the following references about using server-side embedded Java before attempting to use ClientObjects:

[Objects in BBJ](#) (PowerPoint)

[Exponentially Better Applications: Embedded Java and BBJ](#) (HTML)

Syntax of ClientObjects

[Back to top](#)

There are three ways in which the @ symbol is used in BBJ to indicate that embedded Java is to use ClientObjects; when

- creating a new ClientObject,
- invoking a static method on a client-side class and
- declaring a value to be a ClientObject.

In each case, the @ is appended to the class name. The class name can be a fully qualified class name or it can be an abbreviated class name if the fully qualified class name has appeared in a [USE](#) statement.

```
00010 use java.util.HashMap
00020 use java.util.Calendar

00030 REM declaring variables to have ClientObject types
00040 declare HashMap@ map!
00050 declare Calendar@ calendar!
00060 declare java.util.ArrayList@ list!
00070 declare java.text.DateFormat@ dateFormat!

00080 REM creating new instances of ClientObjects
00090 map! = new HashMap@()
00100 list! = new java.util.ArrayList@()

00110 REM invoking static methods on client-side classes
00120 calendar! = Calendar@.getInstance()
00130 dateFormat! = java.text.DateFormat@.getDateTImeInstance()
```

Code Sample 3: The various usages of the @ within a BBJ program

At lines 040-070 a number of variables are declared to be ClientObjects. At lines 090-110 two client-side objects are created and assigned to ClientObject variables. At lines 120-130, two static methods are called on client-side classes and the return values of those static methods are assigned to ClientObject variables.

Using a ClientObject

[Back to top](#)

The methods of a ClientObject are called in the same way that the methods of other objects are called; by using the standard dot notation. In general, the parameters that are passed when calling a method on a ClientObject must themselves be ClientObjects. If a method accepts a string or a number then it will accept a server-side value (see following section for details).

The return value of a method called on a ClientObject is again generally a ClientObject unless that return value is a number or a string. If the return value is a number or a string then the return value will be a server-side value.

```
0010 use java.util.HashMap
0020 use java.util.Calendar

0030 declare HashMap@ map!
0040 declare Calendar@ calendar!

0050 map! = new HashMap@()
0060 calendar! = Calendar@.getInstance()
0070 print calendar!
0080 print
0090 print calendar!.getTime()
0100 print

0110 map!.put("calendar",calendar!)
0120 map!.put(123, 456)

0130 print map!.get("calendar")
0140 print
0150 print map!.get(123)
```

Code Sample 4: Calling methods on ClientObjects

Running code sample 4 results in output similar to the sample below.

```
[CLIENTOBJECT] @139f593
java.util.GregorianCalendar[time=1178801482531,areFieldsSet=true,areAllFields
Set
=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/Denver",offset
=-2
5200000,dstSavings=3600000,useDaylight=true,transitions=157,lastRule=java.uti
l.S
impleTimeZone[id=America/Denver,offset=-
25200000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMon
th=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,
endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayO
fWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2007,MONTH=4,WEEK_OF_YEAR=19,WEEK
_OF_MONTH=2,DAY_OF_MONTH=10,DAY_OF_YEAR=130,DAY_OF_WEEK=5,DAY_OF_WEEK_IN MONT
H=2,AM_PM=0,HOUR=6,HOUR_OF_DAY=6,MINUTE=51,SECOND=22,MILLISECOND=531,ZONE_OFF
SET=-5200000,DST_OFFSET=3600000]

[CLIENTOBJECT] @1976073
Thu May 10 06:51:22 MDT 2007

[CLIENTOBJECT] @139f593
```

```

java.util.GregorianCalendar[time=1178801482531,areFieldsSet=true,areAllFields
Set
=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/Denver",offset
=-2
5200000,dstSavings=3600000,useDaylight=true,transitions=157,lastRule=java.uti
l.S
impleTimeZone[id=America/Denver,offset=-
25200000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMon
th=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,
endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayO
fWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2007,MONTH=4,WEEK_OF_YEAR=19,WEEK
_OF_MONTH=2,DAY_OF_MONTH=10,DAY_OF_YEAR=130,DAY_OF_WEEK=5,DAY_OF_WEEK_IN_MONT
H=2,AM_PM=0,HOUR=6,HOUR_OF_DAY=6,MINUTE=51,SECOND=22,MILLISECOND=531,ZONE_OFF
SET=-5200000,DST_OFFSET=3600000]
456

```

Output of Sample 4

The toString() value of a ClientObject consists of the string "[CLIENTOBJECT]" followed by an ID followed by the toString() value of the client-side object.

Notice that looking at the output of sample 4, calendar! is a ClientObject and that the return value of calendar!.getTime() is also a ClientObject. If we place calendar! into a client-side HashMap and then retrieve it, we will receive the same ClientObject. And if we place the server-side number 456 into the client-side HashMap and then retrieve it, we will receive the server-side number.

The field values of a client-side object can also be accessed through the ClientObject and the static fields of a client-side class can be accessed by addressing that class using the @ symbol.

```

0010 use java.awt.Color
0020 declare Color@ blue!
0030 blue! = Color@.BLUE
0040 print blue!

```

Code Sample 5: Accessing fields of client-side objects and static fields of client-side classes

ClientObject vs. Server-side Objects

[Back to top](#)

It is important to understand the difference between a ClientObject and a server-side Object. Any Object that was created using the '@' notation is a ClientObject. Any Object that was created without the '@' notation is a server-side Object. BBjAPI is a server-side Object. Any Object that is obtained through calls on BBjAPI is a server-side Object. In particular, all BBjControls are server-side Objects even though they 'represent' GUI Objects that exist on the client.

In general, the parameters passed to a ClientObject must be ClientObjects while the parameters passed to a server-side object must be server-side objects.

The exceptions to this general rule are:

- 1) Strings and Numbers are always server-side values. Strings and Numbers may be passed as parameters to methods of ClientObjects.
- 2) The method [BBjWindow::addWrappedJComponent](#) (which is a method call on a server-side object) accepts as a parameter a ClientObject.

- 3) A BBJControl (which is a server-side object) can be passed as a parameter to a ClientObject if that ClientObject represents a client-side swing component.
- 4) A BBJ CustomObject (which is server-side object) can be registered as an event listener on a ClientObject that represents a swing component

These exceptions allow a program to place objects that extend javax.awt.JComponent@ (ie any client-side JComponent) onto a [BBJWindow](#) as well as to place [BBJControls](#) onto JComponents and to respond to events that occur on JComponents. All these use cases are demonstrated in the following sections.

Similarly, the return value of a method invocation on a ClientObject is itself a ClientObject and the return value of a method invocation on a server-side Object is a server-side Object except that:

- 1) Strings and Numbers are always server-side values.
- 2) [BBJBarChart](#), [BBJLineChart](#), and [BBJPieChart](#) (which are server-side Objects) each have a method getClientChart() which returns a ClientObject.

Placing a JComponent into a BBJWindow

[Back to top](#)

BBJ 8.0 provides a new method BBJWindow::addWrappedJComponent() that allows the program to create a BBJControl that contains a client-side Java Component. In order to place a client-side JComponent onto a BBJWindow, the program first creates a ClientObject that represents the JComponent and then 'wraps' the Object in a [BBJWrappedJComponent](#) by calling [addWrappedJComponent](#).



The following code creates a JButton that has a thick green border and places it onto a BBJWindow as shown to the right.

```
0010 use javax.swing.border.LineBorder
0020 use java.awt.Color
0030 use javax.swing.JButton

0040 REM declare server side variables
0050   declare BBJSysGui sysGui!
0060   declare BBJWindow window!
0070   declare BBJControl xmasButton!

0080 REM declare ClientObject variables
0090   declare Color@ green!
0100   declare LineBorder@ border!
0110   declare JButton@ jButton!

0120 REM create a window
0130   sysGui! = BBJapi().openSysGui("X0")
0140   window! = sysGui!.addWindow(50,50, 100, 100,
0140: "custom button demo")

0150 REM create a ClientObject that represents a client side
0160 REM JButton with a thick green border
0170   green! = Color@.GREEN
```

```

0180 border! = new LineBorder@(green!, 10)
0190 jButton! = new JButton@()
0200 jButton!.setBorder(border!)

0210 REM add ClientObject as a wrapped component on the window
0220 xmasButton! = window!.addWrappedJComponent(101, 10,10, 80,
220: 60, jButton!)

0230 REM manipulate xmasButton! as a BBJControl
0240 xmasButton!.setText("hello")
0250 xmasButton!.setLocation(35,15)

0260 escape

```

Code Sample 5: Placing a JButton onto a BBJWindow

Registering a BBJ Callback as a Listener on a JComponent [Back to top](#)

A BBJ CustomObject may be registered as an event listener on a ClientObject by calling the add<some>Listener method of the ClientObject.

The CustomObject must have methods that 'correspond' to the methods of <some>Listener. Each of the corresponding methods of the CustomObject must have the same name as the method of the <some>Listener and must accept ClientObjects where the <some>Listener accepts events.

So, for example, the following code sample registers a CustomObject named Listener as a PropertyChangeListener on a JSplitPane. A PropertyChangeListener has a single method

void PropertyChange(PropertyChangeEvent event)

so the CustomObject, Listener, must have a corresponding method

void PropertyChange(PropertyChangeEvent@ event)

with the same name but accepting a PropertyChangeEvent@ rather than a PropertyChangeEvent.

```

0010 use java.beans.PropertyChangeEvent
0030 use javax.swing.JPanel
0040 use javax.swing.JSplitPane
0050 use java.awt.Dimension

0060 declare BBJSysGui sysgui!
0070 declare BBJWindow window!
0080 declare JSplitPane@ splitPane!
0090 declare JPanel@ panel!
0100 declare Listener listener!

0110 REM create a window
0120 width = 500
0130 height = 400
0140 open (unt)"X0"
0150 sysgui!=BBJapi().getSysGui()
0160 window! = sysgui!.addWindow(10,10,width, height,"BBJWindow")

0170 REM add a JPanel as a wrappedComponent
0180 panel! = new JPanel@()
0190 window!.addWrappedJComponent(2222,0,0,width, height, panel!)

0200 REM create a JSplitPane and place it into the JPanel

```

```

0210 splitPane! = new JSplitPane@()
0220 dimension! = new Dimension@(width, height)
0230 splitPane!.setPreferredSize(dimension!)
0240 splitPane!.setContinuousLayout(1)
0250 orientation = javax.swing.JSplitPane.HORIZONTAL_SPLIT
0260 splitPane!.setOrientation(orientation)
0270 panel!.add(splitPane!)
0280 panel!.validate()

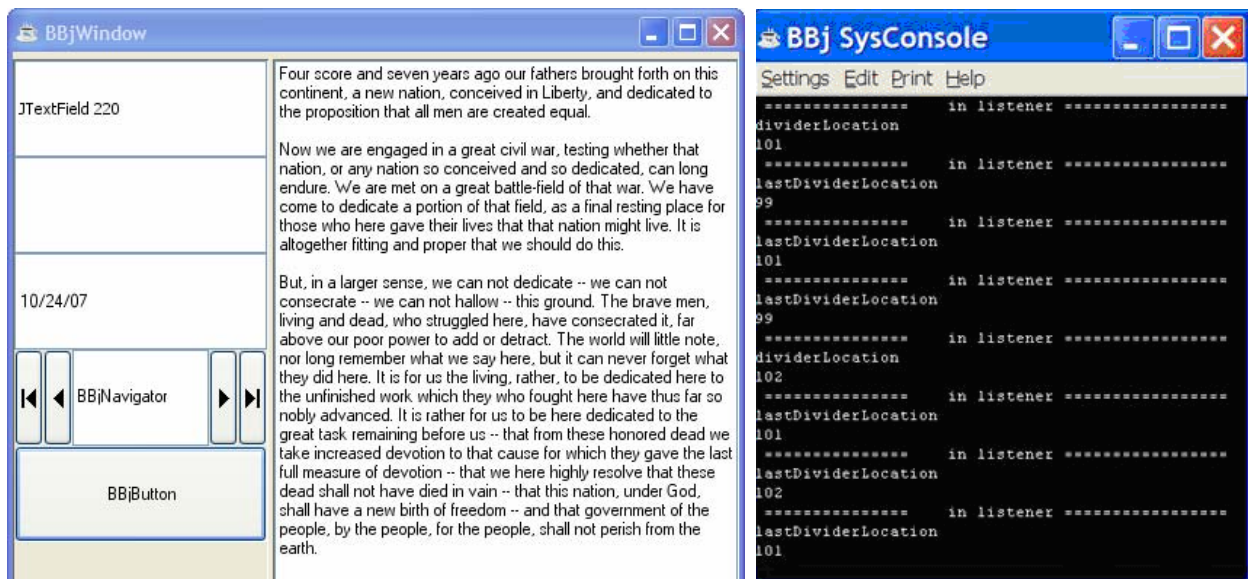
0290 REM create a CustomObject and register it as a
0300 REM PropertyChangeListener on the JSplitPane
0310 listener! = new Listener()
0320 splitPane!.addPropertyChangeListener(listener!)

0330 process_events

0340 class public Listener
0350     method public void propertyChange(
0350:         PropertyChangeEvent@ p_event!)
0360     print " ===== in listener ====="
0370     print p_event!.getPropertyName()
0380     print p_event!.getNewValue()
0390     methodend
0400 classend

```

Code Sample 6: Adding a JSplitPane to a BBJWindow and registering a listener



Output of Code Sample 6

Console Output of Sample 6: As the separator of the JSplitPane is moved

Mixing ClientObjects and BBJControls to Provide a BBJWindow with a LayoutManager [Back to top](#)

We are now in a position to build a program that provides an interesting mix of JComponents and BBJContols. The following sample provides a BBJWindow that contains a JSplitPane. The left side of the JSplitPane contains both BBJControls and JComponents. In addition, the left

panel has a layout manager so that all the controls (both the JComponents and the BBjControls) change sizes as the separator of the JSplitPane is moved.

Code Sample 7: A mix of JComponents and BBjControls with a LayoutManager

```
0010 use java.beans.PropertyChangeEvent
0030 use javax.swing.JPanel
0040 use javax.swing.JSplitPane
0050 use javax.swing.JFormattedTextField
0060 use java.awt.Dimension
0070 use java.awt.GridLayout
0090 use java.awt.event.FocusEvent

0100 declare BBjSysGui sysgui!
0110 declare BBjWindow window!
0120 declare BBjButton button!
0130 declare JSplitPane@ splitPane!
0140 declare JPanel@ panel!
0150 declare Listener listener!

0160 REM open sysgui and add a window
0170   sysgui!=BBjapi().openSysGui("X0")
0180   window! = sysgui!.addWindow(10,10,500,400,
180:     "BBjWindow", $00010003$)
0190   panel! = new JPanel@()

0200 REM create some BBjControls
0210   inputn! = window!.addInputN(102,10,10,90,30)
0220   inputd! = window!.addInputD(103,10,10,90,30)
0230   navigator! = window!.addNavigator(104,10,10,90,30,
230:     "BBjNavigator")
0240   button! = window!.addButton(1,10,10,90,30,"BBjButton")

0250   gosub gettysburg
0260   cedit! = window!.addCEdit(106,10,10,200,300,gettysburg$, $0002$)

0270 REM add a wrapped JPanel to the BBjWindow and
0280 REM place a JSplitPane into that JPanel
0290   window!.addWrappedJComponent(2222,0,0,500,400, panel!)
0300   splitPane! = new JSplitPane@()
0310   dimension! = new Dimension@(window!.getWidth(),
0310:     window!.getHeight())
0320   splitPane!.setPreferredSize(dimension!)
0330   splitPane!.setContinuousLayout(1)
0340   splitPane!.setOrientation(javax.swing.
0340:     JSplitPane@.HORIZONTAL_SPLIT)

0350 REM create a JPanel that will become the left side panel
0360 REM give it a GridLayout
0370   layout! = new GridLayout@(6,1)
0380   leftPanel! = new JPanel@(layout!)

0390 REM add a JComponent to the left panel
0400   jEdit! = new JFormattedTextField@()
0410   jEdit!.setText("JTextField 220")
```

```

0420  jEdit!.setBounds(10,10,90,30)
0430  jEdit!.setVisible(1)
0440  leftPanel!.add(jEdit!)

0450 REM place most of the BBJComponents into the left panel
0460  leftPanel!.add(inputn!)
0470  leftPanel!.add(inputd!)
0480  leftPanel!.add(navigator!)
0490  leftPanel!.add(button!)

0500 REM place the remaining BBJComponent into the panels
0510  splitPane!.add(leftPanel!, javax.swing.JSplitPane@.LEFT)
0520  splitPane!.add(credit!, javax.swing.JSplitPane@.RIGHT)

0530 REM place the JSplitPane into the wrapped JPanel
0540 REM and validate the wrapped JPanel
0550  panel!.add(splitPane!)
0560  panel!.validate()

0570 REM create a listener. Register it to listen for
0580 REM PropertyChangeEvents on the splitPane and for
0590 REM FocusEvents on the jEdit
0600  listener! = new Listener()
0610  splitPane!.addPropertyChangeListener(listener!)
0620  jEdit!.addFocusListener(listener!)

0630 REM set some BBJ callbacks
0640  window!.setCallback(window!.ON_CLOSE,"eoj")
0650  window!.setCallback(window!.ON_RESIZE,"resize")
0660  button!.setCallback(button!.ON_BUTTON_PUSH,
0660:      listener!,"buttonPush")

0670 process_events

0680  eoj:
0690  release

0700  resize:
0710  event! = sysgui!.getLastEvent()
0720  dimension! = new Dimension@(event!.getWidth(),
0720:      event!.getHeight())
0730  splitPane!.setPreferredSize(dimension!)
0740  return

0750 class public Listener
0760  method public void propertyChange(
0760:      PropertyChangeEvent@ p_event!)
0770  print " ===== in listener ====="
0780  target! = p_event!.getSource(); print "target: ", target!
0790  print p_event!.getPropertyName()
0800  print p_event!.getNewValue()
0810  methodend

0820  method public void buttonPush(BBJButtonPushEvent event!)
0830  print "button pushed"
0840  methodend

```

```

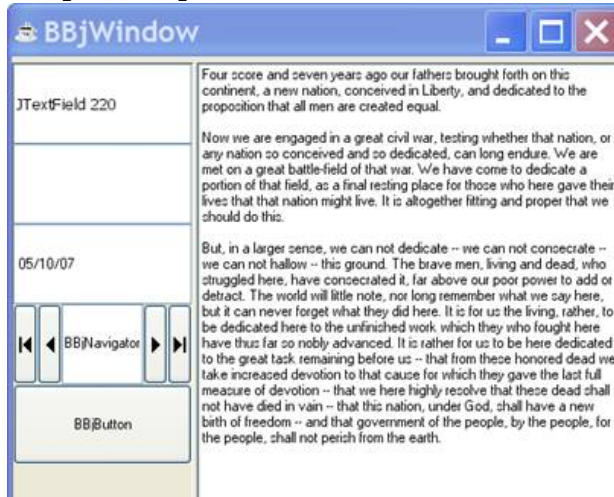
0850     method public void focusGained(FocusEvent@ p_event!)
0860         print "GainedFocus:"
0870         print p_event!.getSource()
0880     methodend

0890     method public void focusLost(FocusEvent@ p_event!)
0900         print "LostFocus:"
0910         print p_event!.getSource()
0920     methodend
0930 classend

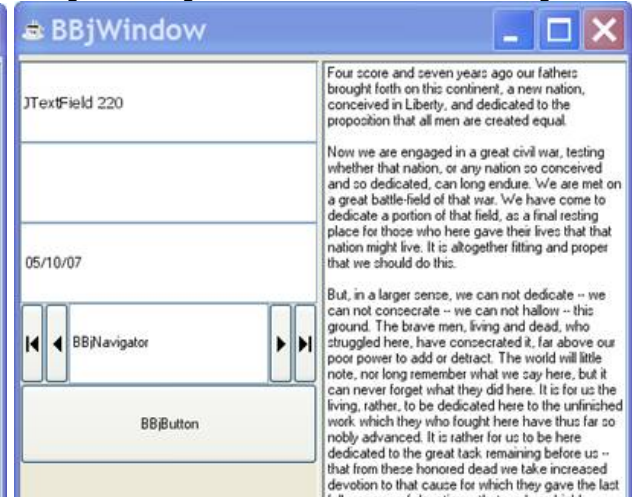
0940 gettysburg:
0950 gettysburg$ = "Four score and seven years ago our fathers brought
forth on this continent, a new nation, conceived in Liberty, and dedicated to
the proposition that all men are created equal."+$0a$+$0a$+ "Now we are
engaged in a great civil war, testing whether that nation, or any nation so
conceived and so dedicated, can long endure. We are met on a great battle-
field of that war. We have come to dedicate a portion of that field, as a
final resting place for those who here gave their lives that that nation
might live. It is altogether fitting and proper that we should do
this."+$0a$+$0a$+ "But, in a larger sense, we can not dedicate -- we can not
consecrate -- we can not hallow -- this ground. The brave men, living and
dead, who struggled here, have consecrated it, far above our poor power to
add or detract. The world will little note, nor long remember what we say
here, but it can never forget what they did here. It is for us the living,
rather, to be dedicated here to the unfinished work which they who fought
here have thus far so nobly advanced. It is rather for us to be here
dedicated to the great task remaining before us -- that from these honored
dead we take increased devotion to that cause for which they gave the last
full measure of devotion -- that we here highly resolve that these dead shall
not have died in vain -- that this nation, under God, shall have a new birth
of freedom -- and that government of the people, by the people, for the
people, shall not perish from the earth."+$0a$
0960 return

```

Sample 7 Output



Sample 7 Output: After user moved the separator



In addition to managing the Layout of the left panel, Sample 7 also reports FocusGain and FocusLoss whenever the JTextField gains or loses focus and reports PropertyChangeEvents whenever the user moves the separator of the JSplitPane.

Requirements for Class files Used to Create ClientObjects [Back to top](#)

In order to create a ClientObject the Java Class file that defines the client-side Object must be visible to the server as well as the client.

In addition, if the Java Class file is not found in a 'privileged' jar file, then BBJ will generate a 'nag' message unless the server is running with a DVK license. A jar file is privileged if it is part of the JRE (Java Runtime Environment) or if it is a 'registered' jar.

The charts.jar file shipped with BBJ is a registered jar file. Developers can register their own jar files using the static Java method `com.basis.jarRegistrationService.client.JarRegistrar.registerJar`.

Code Sample 8: Registering a Jar within a BBJ program

```
USE com.basis.jarRegistrationService.client.JarRegistrar
inputJar$ = "someUnregisteredJar.jar"
outputJar$ = "registered.jar"
JarRegistrar.registerJar(inputJar$, outputJar$)
```

For a full description of JarRegistration and the [JarRegistrar](#) class.

Once a jar file has been registered, it can also be signed without invalidating the BBJ registration. However, any change to the files within the jar will invalidate the registration.

ClientObjects and Java Permissions [Back to top](#)

There are several Java Permissions that are specifically not granted to ClientObjects. In particular, ClientObjects do not have the following permissions:

```
RuntimePermission("setSecurityManager")
RuntimePermission("exitVM")
SQLPermission("setLog")
SocketPermission(*) for ports under 1024
RuntimePermission("createClassLoader")
RuntimePermission("setContextClassLoader")
```

Any attempt to executed ClientObject code that requires one of these Java Permissions will generate a !ERR=18.