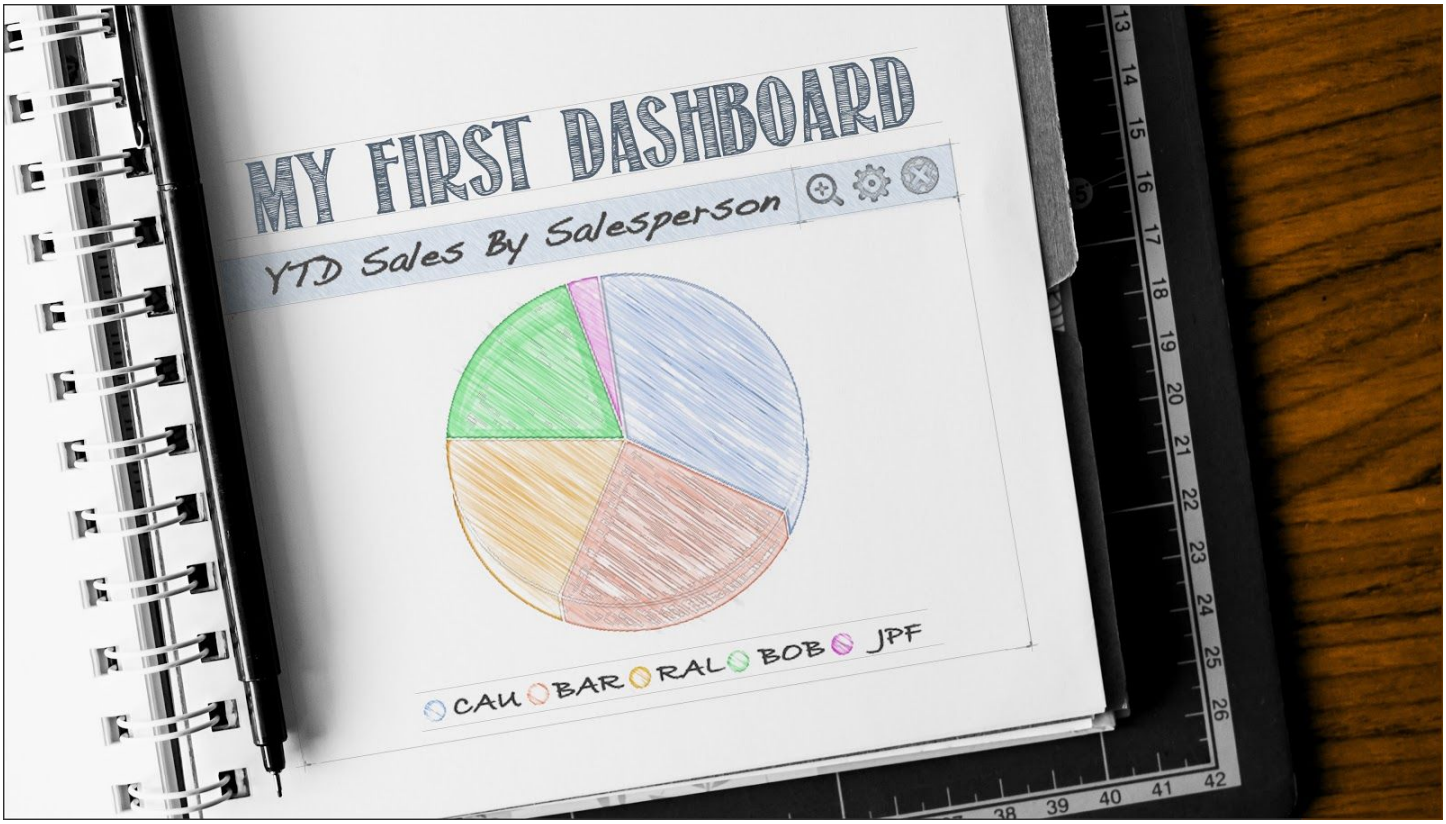


Getting Started with the Dashboard Utility

By Nick Decker

The Dashboard Utility Framework Overview

Before we start digging into the various widget types and extracting data, it is a good idea to take a step back and familiarize ourselves with the Dashboard Utility as an object-oriented framework. Without going into too many of the gory details, the Dashboard Utility is a set of BBj® programs that use BBj Custom Objects to build complex objects via classes. The final bit of jargon is that the Dashboard Utility adheres to the Model–View–Controller (MVC) architectural pattern, which is useful to keep in mind because most of the time we will be dealing with BBj objects that represent a particular model when writing dashboard programs. Various objects, like the Dashboard and DashboardWidget, are models. Wikipedia defines models as the application's dynamic data structure, independent of the user interface. The models directly manage the data, logic, and rules of the application and objects without exposing the underlying controls or details. In future articles, we will dig deeper into the differences between the model and view objects and how your program interacts with them. If all these terms are new to you, take comfort that they make life a lot simpler for the Business BASIC programmer by reducing complexity and providing high-level synergy with dashboard objects. We will cover some of the theory and background of BBj Custom Objects in this article, but you will not need to fully understand all the underlying concepts, because the resultant BBj program will be small and easy to read. This article focuses on creating a

fully-functional dashboard program in a minimal amount of code, with the Dashboard Utility taking care of the low-level details and code for us.

The Dashboard Program Flowchart

To get a better idea of what our dashboard program will look like, including the main steps and data objects that we will work with, refer to the Dashboard Design Process flowchart in **Figure 1**.

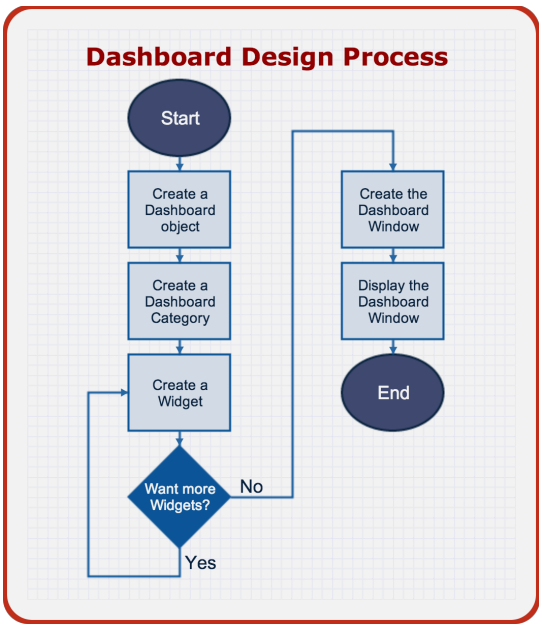
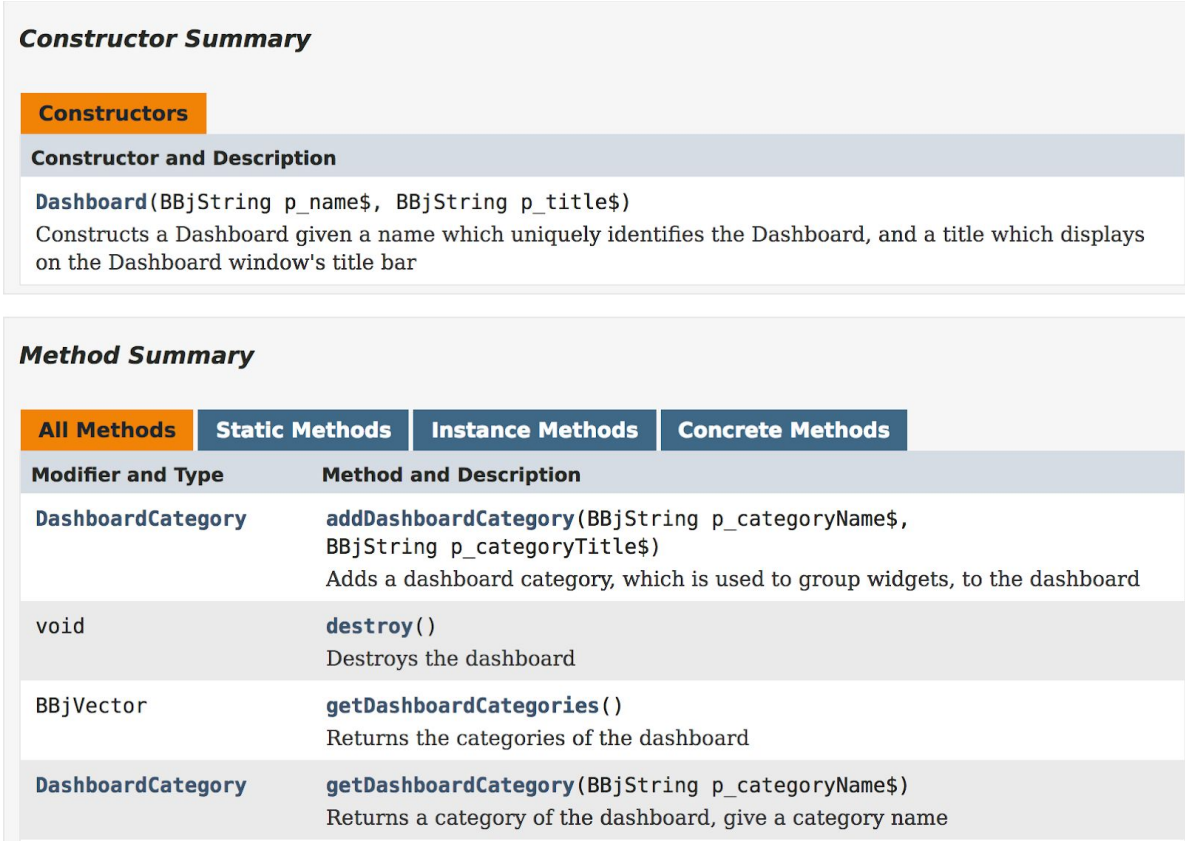


Figure 1. A flowchart of our dashboard program's design process

This will be the standard flow of most dashboard programs, although it is possible to bypass the dashboard framework and create independent widgets such as pie charts and report widgets that you can embed inside an existing BBj application's window. That process is different, and a future article will detail that type of widget development.

Creating a Dashboard Class Instance Via a Constructor

Our program begins by creating a Dashboard object. This is where we will first come in contact with the custom classes, as we instantiate (or create) a new Dashboard object as an instance of the Dashboard class. In Object Oriented programming, sometimes referred to as OOP, you accomplish this via the Dashboard class's constructor. It is worthwhile to mention that BASIS programmatically documents all the Dashboard Utility's classes and their methods via the BBjToJavadoc utility. This utility generates the documentation, which is then published along with the rest of the BASIS Product Suite Help. The end result is that the Dashboard object is fully documented in Javadoc format here, and **Figure 2** shows an excerpt of one of those pages that we will use when writing our first line of code.



Constructor Summary

Constructors

Constructor and Description
`Dashboard(BBjString p_name$, BBjString p_title$)` Constructs a Dashboard given a name which uniquely identifies the Dashboard, and a title which displays on the Dashboard window's title bar

Method Summary

All Methods | **Static Methods** | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`DashboardCategory`	`addDashboardCategory(BBjString p_categoryName$, BBjString p_categoryTitle$)` Adds a dashboard category, which is used to group widgets, to the dashboard
`void`	`destroy()` Destroys the dashboard
`BBjVector`	`getDashboardCategories()` Returns the categories of the dashboard
`DashboardCategory`	`getDashboardCategory(BBjString p_categoryName$)` Returns a category of the dashboard, give a category name

Figure 2. An excerpt of the Dashboard object's Javadoc documentation page

This page lists everything about the Dashboard object, including fields (a variable specific to the class), constructors, and the available methods that provide us with ways to interact with the object after we instantiate it.

Instantiating the Dashboard Object

Earlier we said that the OOP jargon may sound daunting, but that it would end up making our application programming job a lot easier. Creating a Dashboard object serves as a good case in point for this claim. The Javadocs state that to construct a new Dashboard object, we must use the following constructor:

```
Dashboard(BBjString p_name$, BBjString p_title$)
```

Given this, our BBj code (with comments) to create our Dashboard object looks like this:

```
rem Create the dashboard object
dashboard! = new Dashboard("myFirstDB","My First Dashboard")
```

The BBj Custom Objects Tutorial provides details about custom objects, constructors, and object variables, and mentions that the exclamation mark (`!`) following a variable indicates that it is an object variable. This is similar to how string variables in BBx have a **$** suffix and integer variables have a **%** suffix. So our line of BBj code constructs a Dashboard object, passing in string values for the dashboard's name and title. BBj assigns the resultant instance to our `dashboard!` object variable, which we will use in upcoming code. While we have not yet displayed anything on the computer screen, when BBj runs this one line of code it initiates the execution of a cascade of Dashboard

Utility code—code that provides a lot of functionality which we will soon cover, but more importantly, code that we did not have to write!

If we were to cheat a bit by jumping ahead and displaying the visual representation of our `dashboard!` object, we would see something like the window shown in **Figure 3**.

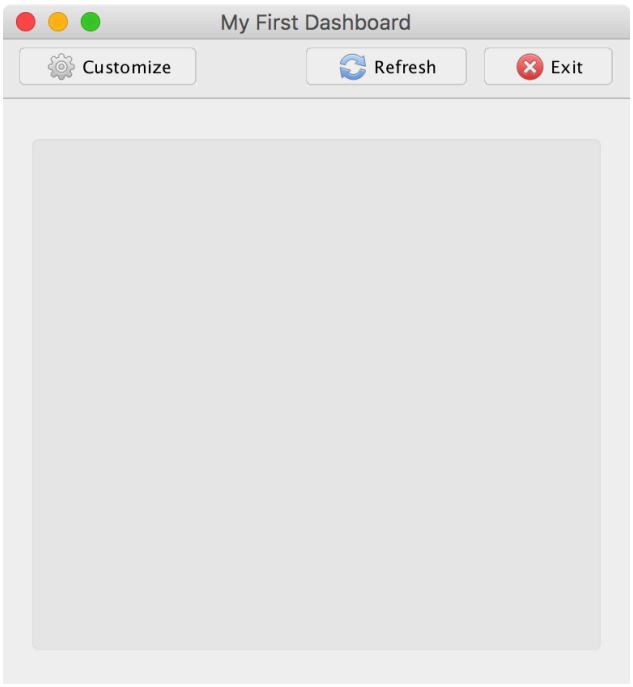


Figure 3. Displaying our `dashboard!` object in a DashboardWindow

Now that we have a concrete Dashboard object, we can call its methods to perform specific tasks. The next step in our flowchart says that we must create a DashboardCategory, which we will do in the next section by calling a method on our new `dashboard!` object variable.

Adding a DashboardCategory to the Dashboard Object

The Dashboard Utility renders a DashboardCategory as a tab inside the dashboard that serves as a container for widgets. It is possible to create many categories, or tabs filled with widgets, in the same dashboard. Generally speaking, categories exist to group widgets that report on similar data. So it is possible to create one category with widgets that focus on sales statistics, and another that deals with accounting metrics. The categories that you create and how you group the widgets into those categories is entirely up to you—it depends on the data you want to present to the end user and how that data is best grouped.

Instantiating the DashboardCategory Object

The DashboardCategory's Javadoc page shows that the object does have a constructor, but that is a testament to the Dashboard Utility's flexibility in that you can create standalone versions of many of the classes. Because we already have our `dashboard!` object we can make use of its `addDashboardCategory()` method as a shortcut to create a category and add it to our existing dashboard in a single step. **Figure 4** is an excerpt

from the Dashboard object's Javadoc page and shows the method that we will use to do this.

Modifier and Type	Method and Description
DashboardCategory	addDashboardCategory(BBjString p_categoryName$, BBjString p_categoryTitle$) Adds a dashboard category, which is used to group widgets, to the dashboard

Figure 4. The Dashboard's `addDashboardCategory()` method documentation

Given that method description, our next line of BBj code looks like this:

```
rem Create the dashboard category to hold our widgets
category! = dashboard!.addDashboardCategory("sales", "Sales Metrics")
```

That line of code executes the **`addDashboardCategory()`** method on our **`dashboard!`** object and provides string values for the category's unique name and the title which displays on the tab. Calling that method gives us a new DashboardCategory object variable called **`category!`**. Here again, we have created a model of a complex custom class and the Dashboard Utility is responsible for creating the associated view, or user interface, for the category. That is another benefit of the MVC paradigm, as our program only deals with a simplified model of the category and the utility goes through the effort of creating the actual tab control and managing selection events and associated child windows.

Jumping ahead once more and displaying what we have created so far results in the window shown in **Figure 5**. The difference between **Figure 5** and the previous screenshot in **Figure 3** is that our dashboard program now displays an empty "Sales Metrics" tab.

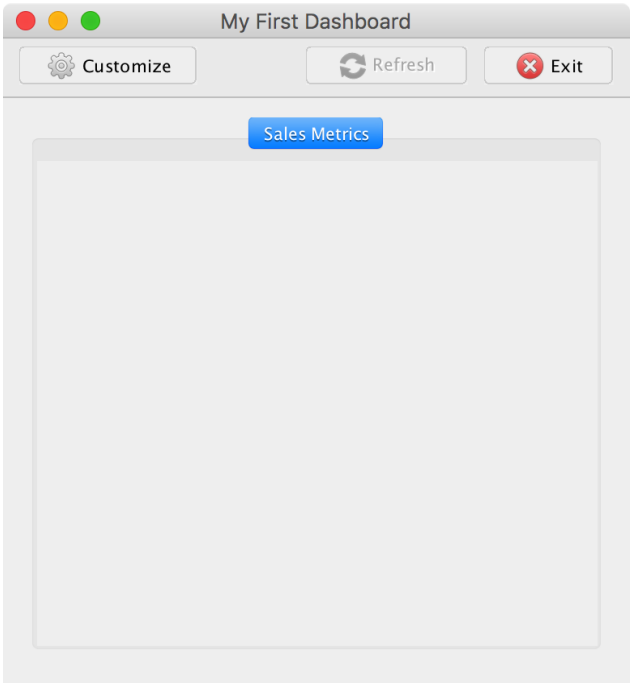


Figure 5. Displaying our "Sales Metrics" **`category!`** object as a tab in the dashboard

Adding Widgets to the DashboardCategory

The next step in our flowchart covers creating widgets and adding them to the DashboardCategory. Earlier we found that we could create a standalone DashboardCategory, but it was easier to call an add method on our **`dashboard!`** object. That is also the case for widgets and our new **`category!`** object. Instead of creating standalone widgets, we will take a shortcut and execute a single method on our **`category!`** object to both create the widget and add it to the category. Looking at the DashboardCategory's Javadoc page, there are about 50 methods that we can execute to add different types of widgets to our category. That is a lot of different ways to add widgets, and it is due to two factors:

1. A large number of available widgets such as various types of charts, grids, reports, images, and HTML views.
2. Several different ways to fill the widget with data, such as SQL result sets, BBjRecordSets, and methods like **`setDataSetValue()`**.

Overloaded Methods

Because we would like to show our sales data for each salesperson as a slice in a pie chart, we will use one of the DashboardCategory's **`addPieChartDashboardWidget()`** methods. This is the place where we will have to provide more code, though, as widgets are flexible enough to allow you to control many aspects of their appearance. In the same way that there is no single BBjTopLevelWindow with a fixed size and position that satisfies every application window need, widgets offer a variety of customization options provided at the time of their creation. Taking that comparison even further, let's look at **Figure 6** which shows the documentation for BBjSysGui methods that create a BBjTopLevelWindow.

Return Value	Method
BBjTopLevelWindow	addWindow(int x, int y, int width, int height, string title)
BBjTopLevelWindow	addWindow(int x, int y, int width, int height, string title, string flags)
BBjTopLevelWindow	addWindow(int x, int y, int width, int height, string title, string flags, string event_mask)
BBjTopLevelWindow	addWindow(int context, int x, int y, int width, int height, string title)
BBjTopLevelWindow	addWindow(int context, int x, int y, int width, int height, string title, string flags)
BBjTopLevelWindow	createTopLevelWindow(int resHandle, int windowID)

Figure 6. The BBjSysGui's methods to create a BBjTopLevelWindow with parameters

It is common for classes to offer many constructors or methods that do the same task, but with a different set of parameters. In OOP parlance, this is known as method overloading. This refers to differentiating the method based on the parameters of the method. As a case in point, there are five distinct versions of the **`addWindow()`** method shown in **Figure 6**. Because they take different parameters, some methods are better suited for particular situations. Some offer very few parameters, which usually indicates that the return object will end up with default values. Other methods give the programmer the ability to fine-tune the window's appearance and behavior at creation time by including parameters for window flags and event masks.

Likewise, when we look at the DashboardCategory's methods we find that there are several different ways of adding a pie chart widget. **Figure 7** shows an excerpt of the available methods that add a pie chart to the category.

DashboardWidget	addPieChartDashboardWidget(BBjString p_name$, BBjString p_title$, BBjString p_previewText$, BBjString p_previewImage$, BBjString p_chartTitle$, BBjNumber p_flat, BBjNumber p_legend, BBjRecordSet p_rs!, BBjVector p_columns!) Creates and returns a pie chart dashboard widget
DashboardWidget	addPieChartDashboardWidget(BBjString p_name$, BBjString p_title$, BBjString p_previewText$, BBjString p_previewImage$, BBjString p_chartTitle$, BBjNumber p_flat, BBjNumber p_legend, BBjString p_connectString$, BBjString p_sql$) Creates and returns a pie chart dashboard widget

Figure 7. Two of the five methods that add a PieChart to a DashboardCategory

Similar to adding a BBjTopLevelWindow to a BBjSysGui object, we can choose between five different methods to add a pie chart to our category. The two methods shown in **Figure 7** use many of the same parameters that are common to all widgets such as a name, title, etc. The difference between the two comes at the end of the parameter list. The top method uses a provided BBjRecordSet to fill the pie chart with data whereas the bottom method uses a JDBC connection and SQL query. In the latter case, the programmer supplies a couple of strings and the Dashboard Utility takes care of making a connection to the database, executing the query, filling a result set, and populating the pie chart with the resultant data. This is a perfect example of what we mean when we say that developers interact with the dashboard objects at a high level and the utility manages the low-level details and code.

Instantiating the DashboardWidget Object

Because the add method takes several parameters, our code first sets those parameter values in named variables to make the code easier to read and maintain, as shown in **Figure 8**.

```
REM Create a PieChartWidget from an SQL query to the Chile Company
name$ = "YTDPie"
title$ = "YTD Sales"
previewText$ = title$
previewImage$ = ""
chartTitle$ = "YTD Sales By Salesperson"
flat = 0
legend = 1
connectString$="jdbc:basis:localhost?database=ChileCompany"
sql$ = "select salesperson, round(sum(sales_ytd),2) as TotalSales " +
:      "from  customer where len(trim(salesperson))>1 " +
:      "group by salesperson order by TotalSales desc"
```

Figure 8. The code that defines the variables used to create the pie chart widget

The variables start with `name$`, which is a unique name for the widget in the category. This will never be visible to the end user but the utility uses it to keep track of the widgets. The name may show up later in a log file if an error occurs in the widget, so it's helpful to provide a name that is meaningful and differentiates this widget from the others. The last two variables, `connectString$` and `sql$`, provide the Dashboard Utility with all the material it needs to fill the pie chart with information from our ChileCompany database. Now that we have set all the parameters, we can add the pie chart DashboardWidget to our category with the code shown in **Figure 9**.

```
dashboardWidget! = category!.addPieChartDashboardWidget(
:       name$,title$,previewText$,
:       previewImage$,chartTitle$,
:       flat,legend,
:       connectString$,sql$)
```

Figure 9. The line of code that creates the pie chart widget and adds it to the category

It is a rather long line of code, so to improve legibility we have used line continuation characters to extend it over multiple lines. Jumping ahead once more, **Figure 10** shows what we have so far after adding our pie chart widget to the dashboard.

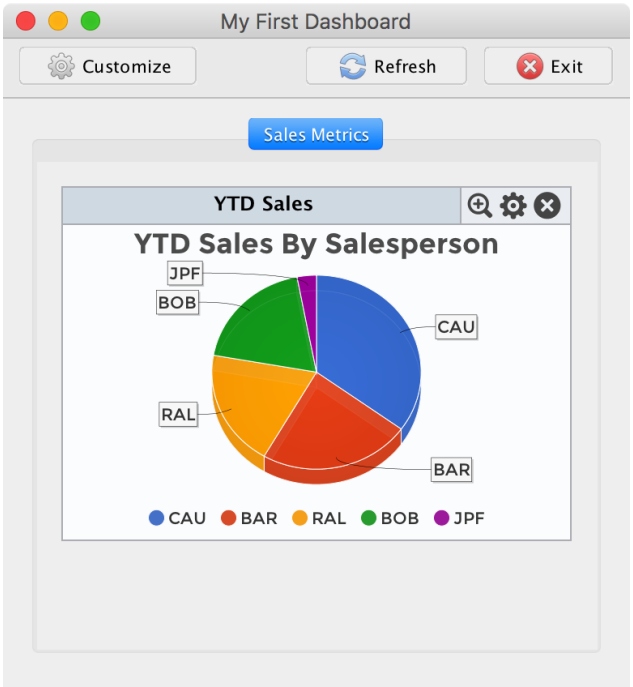


Figure 10. Displaying our pie chart widget in the dashboard

Displaying the Dashboard

As is evident by our sneak peek in **Figure 10**, we have completed most of the code necessary to create a dashboard application. Even though our program is short, there is

little else for us to do before we are finished and can interact with the dashboard. **Figure 11** lists the final lines of code that correspond to the last two items in our flowchart. These lines are responsible for displaying the dashboard and handing control over to the utility to manage events.

```
dashboardWindow! = new DashboardWindow(dashboard!)
dashboardWindow!.doModal()
release
```

Figure 11. The last few lines of code that are responsible for displaying the dashboard

In the first line of code, the `new` operator indicates that we are constructing a DashboardWindow object and passing in our previously-built `dashboard!` object as the only parameter. The Javadocs for the DashboardWindow say that it is the UI class that displays a dashboard in a BBjTopLevelWindow, so this time we are dealing with one of the View portions of the MVC model. The second line of code instructs our new `dashboardWindow!` object to display modally. The Dashboard Utility will handle all the window events for us, so there is no need to write callback code. The utility will return control to our program after the user closes the dashboard window, and our code will then exit courtesy of the `release` verb.

Interacting with the Dashboard and ChartWidget

Figure 10 above shows what our final dashboard looks like with all the code in place, but the framework gives us much more than a pretty pie chart. Dashboard widgets come with a lot of built-in functionality that we get for free, such as the ability to pop the chart out of the dashboard so that it displays in a much larger, user-resizable window as shown in **Figure 12**.

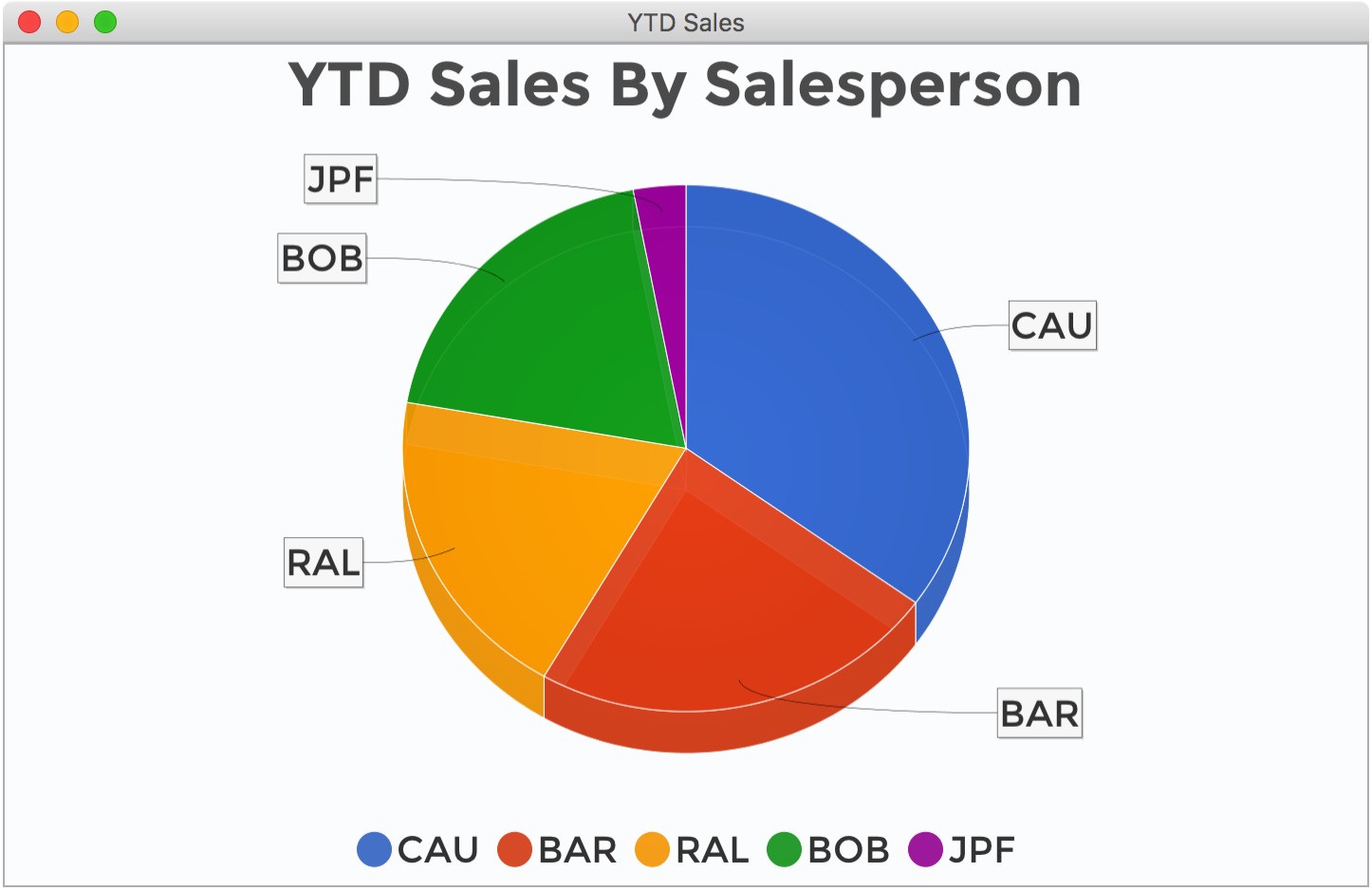


Figure 12. A popped-out version of the widget in a resizable window

Both the dashboard widget and the popped-out version offer the user the ability to do all sorts of things with the widget including:

- Saving out a picture of the pie chart
- Emailing the chart to a colleague
- Exporting the underlying dataset to a CSV file
- Viewing the data in the user's default spreadsheet program, such as Microsoft Excel

Since we filled the widget with the results of an SQL query, by definition it adheres to the RefreshableWidget interface. This is another bonus that the utility provides, as the user can force the chart to update with a snapshot of the most current data in the database. If the widget tracks data that is in a constant state of flux, the developer or the user can configure the widget to repeatedly refresh itself based on a time interval. The interval could be once every 15 seconds, 5 minutes, 2 hours, or whatever delay makes the most sense given the volatility of the data.

All these abilities are available from the widget's Options popup menu, which can be accessed in a couple of different ways. When a DashboardWidget is contained within the dashboard window, it includes a toolbar with the widget's title along with three toolbuttons that permit the user to:

1. Pop out the widget from the dashboard into its own window
2. Display the widget's Options menu

3. Close the widget, effectively hiding it in the category

This same popup menu is also shown when the user right-clicks on the chart, which comes into play for popped-out widgets as they do not display a toolbar. It is worth pointing out that we are describing the widget's default behavior for its Options menu. If the application has specialized needs, the developer can override this behavior and replace the default menu with a custom BBjPopupMenu of their own. It is also possible to register for a right-click event on the widget so that the application can bypass the menu and instead execute custom code of the developer's choosing. **Figure 13** shows the popup menu being accessed by clicking on the Options toolbutton.

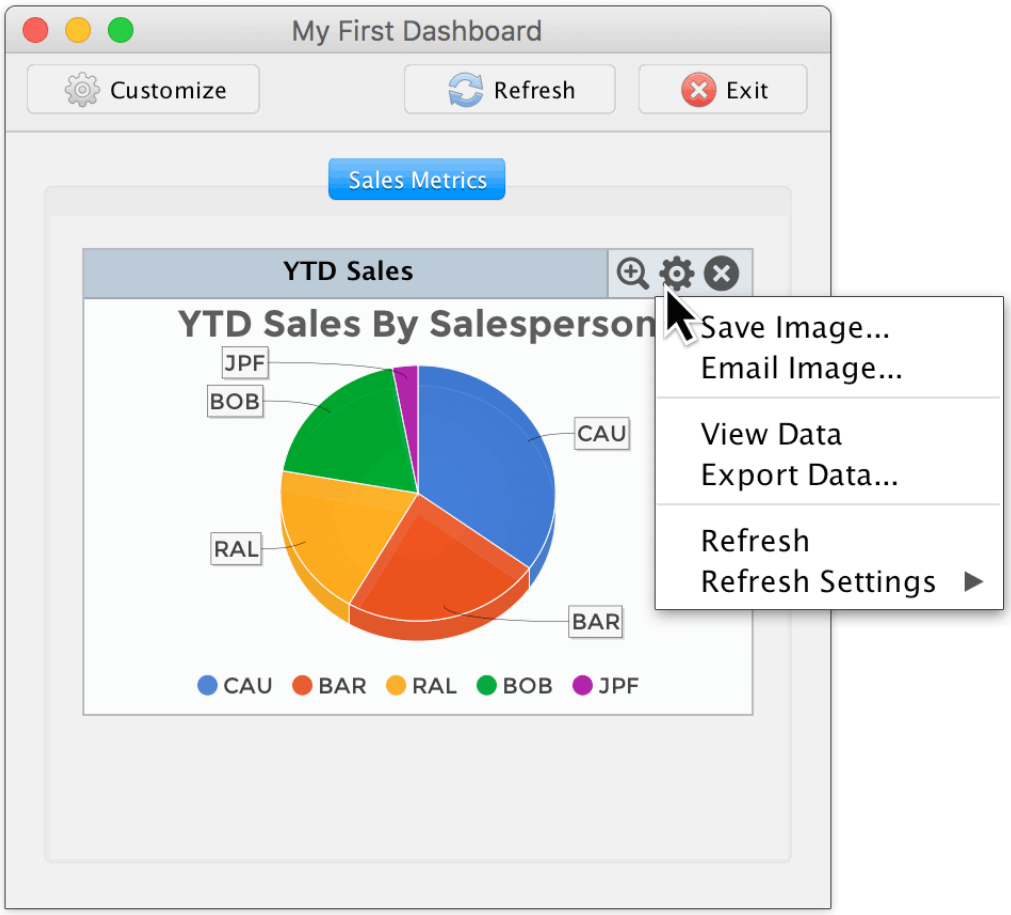


Figure 13. Accessing the widget's Options menu via the toolbutton

Because the Sales Metrics category contains refreshable widgets, the dashboard shows an enabled [Refresh] button at the top of its window. With a single button press, the user can force all widgets in the category to refresh themselves with up-to-date data. Our sneak-peek in **Figure 5** shows the same category without any widgets loaded, so in that screenshot, the utility has disabled the dashboard's [Refresh] button.

Interacting with Chart Tooltips

The user can interact with the chart in other ways as well, such as clicking on a slice in the pie chart or a bar in a bar chart. After clicking on a chart's data entity, the widget displays helpful information about the selected zone in a tooltip, as shown in **Figure 14**.

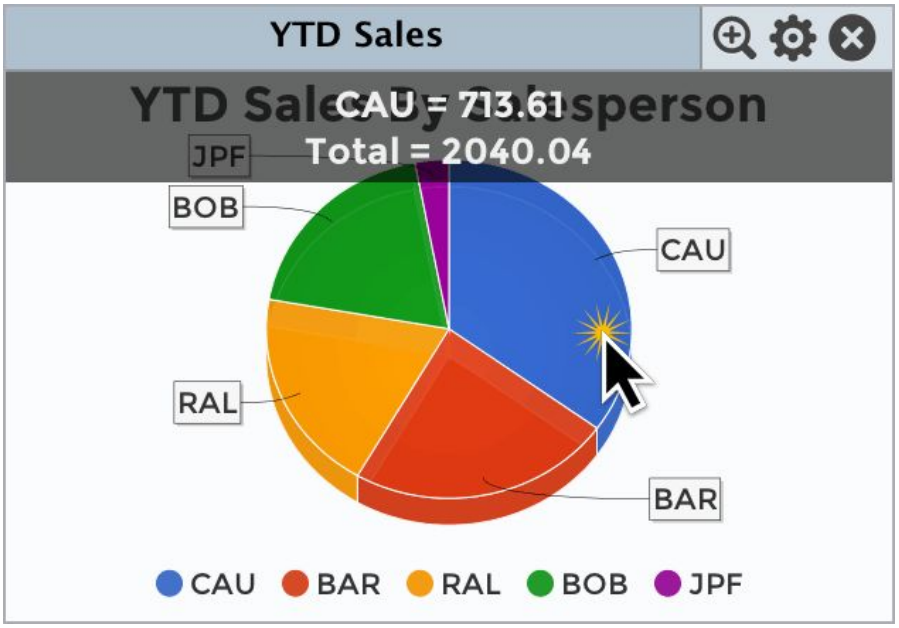


Figure 14. Viewing the data associated with a slice of the pie chart

By default, chart widgets display the data for the selected entity in a translucent tooltip at the top of the chart. **Figure 14** shows that after clicking on the blue slice for our salesperson CAU (Constance Anne Unger), the widget shows that her year-to-date sales are $713.61 and the total for all salespersons is $2,040.04. This is yet another advantage that widgets offer without any extra programming on our part. However, if we were so inclined, we could add code to manipulate the tooltip, including setting the tooltip's background color and opacity, foreground color and opacity, and even the amount of time that the tooltip remains visible before disappearing. If we wanted to exert the utmost control over the tooltip, we can set a callback on the widget and run custom code when the user clicks on the chart. That gives us the opportunity to create our own text to be displayed in the tooltip, or to drill down into the selected sales details and display the relevant orders in a popup GridWidget. We will take a deeper dive into callbacks and custom event handling in a future article.

Object-Oriented Programs and the USE Verb

We mentioned earlier that the Dashboard Utility was a collection of custom BBj classes in BBj program files. Our simple program makes extensive use of these classes and references them to create all of our objects, such as the `dashboard!`, `category!`, and `dashboardWindow!` object variables. But we cannot invoke a new Dashboard object without first telling BBj about the Dashboard class and where the BBj program resides that contains the class definition and code. That is where the USE Verb enters the picture. **Figure 15** shows the `use` statements that we included in our program to make everything work.

```
use ::dashboard/dashboard.bbj::Dashboard
use ::dashboard/dashboard.bbj::DashboardCategory
use ::dashboard/dashboard.bbj::DashboardWidget
use ::dashboard/dashboard.bbj::DashboardWindow
```

Figure 15. Our program's USE statements

Using the double colons as a separator, our `use` statements consist of two main parts: the path to the BBj program that contains the class definition that we want to use and the case-sensitive name of the class. Notice that we did not provide a full path to the **`dashboard.bbj`** program file. This is because the `use` statement takes advantage of prefix entries, and our configuration file includes the Dashboard Utility's parent directory of **`<BBjHome>/utils/`** in our prefix.

Using the USE Verb

Earlier in this article, we covered how we could create an instance of the Dashboard class via a constructor in this line of code:

```
dashboard! = new Dashboard("myFirstDB","My First Dashboard")
```

BBj will only be able to execute that line if we tell it where to find the Dashboard class definition. This is accomplished in the first `use` statement shown earlier in **Figure 15**. If we forgot to include that `use` statement in our program then we would get an error when BBj attempted to execute the line that creates the `dashboard!` object, as shown in **Figure 16**.

```
!ERROR=17  (No type named Dashboard visible in this scope)
[8] dashboard! = new Dashboard("myFirstDB","My First Dashboard")
```

Figure 16. The error that results when we do not include a `use` statement

This concept should not feel too foreign to BBx® programmers, because it is comparable to how the CALL Verb functions. BBx programs can `call` routines that exist in external libraries by specifying the program file and optionally including an embedded label reference. The label in the `call` statement is preceded by two colons (`::`) that specify the starting location in the called program. For example:

```
call "myProgram.bbj::mySubroutine"
```

That line of code is quite similar to our program's first `use` statement:

```
use ::dashboard/dashboard.bbj::Dashboard
```

Both statements specify the location of a BBj program file and either a specific subroutine (defined by the program label for the `call`) or a specific class (delineated by the CLASS and CLASSEND verbs for the `use`). The big difference is that we only have to

include the USE verb *once*, anywhere in our program file, which enables our BBj program to find and reference the Dashboard custom object class.

The last important point about the USE Verb is that we must include a distinct `use` statement for *each* external custom class that we use in our program. That explains why **Figure 15** shows that our program includes four `use` statements - one for each of the Dashboard Utility classes that we create. Our program was rather short and only created a single DashboardWidget, but it is far more common to have a dashboard show multiple widgets in one or more categories. Regardless of the number of DashboardWidgets we create in our code, we only include a single `use` statement for the DashboardWidget class, as its sole purpose is to apprise BBj of the class's definition and file location.

Summary

This article promised to cover some of the theory behind BBj Custom Objects, Object Oriented Programming, and the Dashboard Utility framework. Through the process of creating a simple dashboard program, we reviewed several essential OOP concepts such as instances, constructors, and methods. Those provided background theory as well as helped to explain the syntax of our program. Even if classes, methods, and objects are new to you, the good news is that you do not need a Computer Science degree to write a useful dashboard program. A good analogy is that most of us routinely drive cars and trucks, which are complicated collections of machinery, without understanding any of the details behind the combustion engine. While it is certainly possible to write your own BBj custom class that encapsulates a dashboard widget, most of us will be content to write simple procedural dashboard programs.

Like traditional BBx programs that make use of external code or libraries, our dashboard example program uses custom utility classes to do all the heavy lifting when it comes to executing code. We can create several high-level models in a few lines of code and the Dashboard Utility takes care of building the visual representation of those models. In doing so, it handles the low-level code to create the controls and even defines and executes complementary actions like saving and exporting data from a widget. Because it manages callbacks, events, and dozens of other implementation details for us, our program has the luxury of being concise. In fact, the core of our dashboard program consists of only two constructor calls and three method calls! **Figure 17** shows the source code in the Eclipse IDE for our fully-functional dashboard program with an interactive PieChartWidget.

```
rem Create the dashboard object
dashboard! = new Dashboard("myFirstDB","My First Dashboard")

rem Create the dashboard category to hold our widgets
category! = dashboard!.addDashboardCategory("sales", "Sales Metrics")

rem Create a PieChartWidget from an SQL query to the Chile Company
name$ = "YTDPie"
title$ = "YTD Sales"
previewText$ = title$
previewImage$ = ""
chartTitle$ = "YTD Sales By Salesperson"
flat = 0
legend = 1
connectString$="jdbc:basis:localhost?database=ChileCompany"
sql$ = "select salesperson, round(sum(sales_ytd),2) as TotalSales " +
:      "from  customer where len(trim(salesperson))>1 " +
:      "group by salesperson order by TotalSales desc"

dashboardWidget! = category!.addPieChartDashboardWidget(
:       name$,title$,previewText$,
:       previewImage$,chartTitle$,
:       flat,legend,
:       connectString$,sql$)

rem Create and show the dashboard in a window
dashboardWindow! = new DashboardWindow(dashboard!)
dashboardWindow!.doModal()
release

use ::dashboard/dashboard.bbj::Dashboard
use ::dashboard/dashboard.bbj::DashboardCategory
use ::dashboard/dashboard.bbj::DashboardWidget
use ::dashboard/dashboard.bbj::DashboardWindow
```

Figure 17. Our completed dashboard program

More Dashboard and Widget Information

A variety of material is available online for the BASIS Dashboard Utility, including other Advantage articles, documentation and Javadocs, YouTube videos, and even a section in the IDE User Group Wiki with tutorials and sample code. Many of these resources are linked below for easy access and further study.

Sample Program

- Download or view the sample code used in this article

Related Advantage Articles

- Dash Boredom With the Dashboard Utility
- Maximizing the Power of the Digital Dashboard Widget
- Easier Decision Making With the Dashboard Utility
- AddonSoftware's Digital Dashboard Takes Off
- The Magic of the Widget Wizard

Dashboard Documentation

- Dashboard Utility Overview
- Dashboard Javadoc Reference Documentation
- Dashboard Chart Types
- Dashboard Charts and Datasets
- Dashboard Chart Customization

Dashboard Videos

- 2015 Dashboard Features
- Adding the New Digital Dashboard Utility to Your App
- Dashboards, Meet Drilldowns
- Refreshing DashboardWidget Filters
- Embedding Widgets in Your BBx App
- The New Widget Wizard - Dashboards and Widgets Without Any Code!
- BASIS Invests in Your Application Development Part 1
- BASIS Invests in Your Application Development Part 2
- BASIS Invests in Your Application Development Part 3

Dashboard Section in the IDE User Group Wiki

The Wiki allows developers to share resources and access documentation, sample code, or other development artifacts provided by the community. It contains several sections devoted to the BASIS IDE, the BDT Plug-In, and other applications and utilities such as the Dashboard Utility and BBJasper. Contact BASIS Sales by emailing info@basis.com for an account and instructions on how to get started accessing tutorials and contributing your own code.

- Dashboard Wiki