Table of Contents

Desktop.com API

The Desktop.com API (DTAPI) offers developers a robust programming language, designed to create compelling web-based applications.

The DTAPI is based on JavaScript, but expands it to include the more robust features of an object oriented programming language, including a class-based object framework with inheritance, and an application package management system. These features supply both ease of use in designing and writing applications as well as faster server access while running the application.

As a web-based design tool, the DTAPI allows extensive server access, and offers a suite of GUI elements. Server access allows developers to manipulate server based user files, including persistent user data, as well as to create extensible applications using programming packages, stored on the server. The DTAPI suite of GUI elements includes interactive elements and graphics enablers. The suite provides tools for component-based design, automated user-interactivity, and widgets for commonly used elements, such as scroll bars and user-input devices.

DTAPI offers:

- support for familiar JavaScript and DHTML programming languages
- class based object framework with inheritance, functions, and methods
- package management system for efficient downloading of applications
- ability to store persistent data
- ability to browse, create, manipulate, upload, download and share server based user files and folders
- graphical user interface builder, including:
    - windows and focus control
    - menus
    - layout managers
    - interactive elements, including event handlers and a suite of widgets:
        - including text input, buttons, check box, list box, slider and scroll bars
    - prompts, questions and other common dialog boxes
    - pane based help window
    - image displays
    - text displays and fonts
- ability to launch applications in single or multiple instances
- type registry for file type mapping
- suite of data types
- browser window launch
- application icons

Developers' Guide

This manual also includes a Developers' Guide, which describes the principles behind the Desktop.com API, and offers an overview of the process used in developing and submitting an application for inclusion on the Desktop.com site.

The Developers' Guide is designed to be used in conjunction with the Desktop API Reference Manual. While the Developers' Guide covers overriding principles and procedures used in implementing the DTAPI, the API Reference Manual defines all packages, classes, methods, and functions used in designing for the Desktop.com platform. Both manuals often make reference to related information in the other.

Devtool

Devtool, Desktop.com's developer tool, provides a unified interface for functions needed to edit and test an application online. Devtool also offers tools specific to the Devtop environment, such as the ability to upload and download files to and from our servers. Developers may write applications off-line, then log into Devtop to upload, compile, run, debug, and touch up their programs. Devtop allows you to create and share applications from one location.

Devtool provides the ability to:

- upload and download code to your account
- compile JavaScript for execution under the Desktop environment
- execute application code
- assemble application projects consisting of multiple files
- edit code on-line
- view debugging output from apps
- access the console, Desktop's command-line interface

Uploading and Downloading Code

Devtool includes its own file storage and management system. Uploading and Downloading code is as easy as clicking on a button, allowing you to store work in progress, as well as completed applications, in one directory, accessible from any internet-connected computer.

Compiling Code

Devtool takes your application code, parses it for errors, and formats it for the Desktop environment. It reports syntax errors, undeclared symbols, and unused symbols. The changes it makes include stripping whitespace and comments, translating blocking methods, and renaming identifiers.

Editing Code

Devtool includes an editor, Edwin, suitable for making quick fix modifications to code on-line. The ability to edit on-line allows developers to debug without uploading and downloading each time a fix is necessary.

The Console

The Console is Desktop's command-line interface, similar to a DOS or Unix shell, which allows you to manipulate files in your filesystem, run applications, execute JavaScript commands, and more. To access the list of commands available, simply launch the Console, type "help," and press the Enter key.

System Requirements

This is a list of the minimum requirements for optimal performance of both the Desktop.com platform and its development environment.

- Pentium II 300mhz or higher
- Windows 98 or NT
- 64 MB RAM or higher
- Microsoft Internet Explorer 5 or later
- 56Kbps modem

(Support for Netscape Navigator is planned for the near future.)

Developers' Guide

The Desktop.com Developers' Guide describes the principles behind the Desktop.com API, and offers an overview of the process used in developing and submitting an application for inclusion on the Desktop.com site.

The Developers' Guide is designed to be used in conjunction with the Desktop API Reference Manual. While the Developers' Guide covers overriding principles and procedures used in implementing the DTAPI, the API itself serves as a reference manual in which all packages, classes, methods, and functions used in designing for the Desktop.com platform are defined. Both manuals often make reference to related information in the other.

The DTAPI Class-Based Object Framework

Most popular object-oriented languages, like C++ and Java, are *class-based*. In class-based languages, the programmer declares *classes* which encapsulate data, and *methods* which act upon or work with that data. Many class-based languages also have the concept of *inheritance*, which allows the programmer to declare a new class that specializes or modifies the behavior of another class.

The DTAPI uses JavaScript's ability to access underlying language components to implement these features natively.

Packages

Desktop.com client code is downloaded to the browser only as it is needed. To make this possible, the client code is broken into *packages*.

Packages are the way the DTAPI encapsulates specific bits of functionality. Each package *exports* particular classes and/or functions, which makes them visible from outside the package. The package may define other classes or functions that are not exported.

A package is a function in which classes, methods, and other functions are declared. Each package has a `Requires` variable, which is named after the function itself. This variable must be an array that lists the names of all packages required for the given package to work. The DTAPI PackageManager will ensure that the required packages are present before it loads the package.

For example, to create a package called `PkgRectangle` which requires a package called `PkgShape`:

```
function PkgRectangle()
     {
          ...functions and classes...
     }

PkgRectangle_Requires = ["PkgShape"];
```

Functions, classes and objects may be exported from the package using the DTAPI `Export` method on the package. Other packages will only have access to the classes and methods that are exported. Note that all of a class' methods may be accessed through the class, so it is not necessary to export them.

For example, to export the function `ILikeRectangles` from the `PkgRectangle` package:

```
function ILikeRectangles()
     {
          ...
     }
PkgRectangle.Export(ILikeRectangles);
```

Classes

Classes define an object with set properties and actions, and may be acted upon by methods defined within the class.

Classes in JavaScript are defined by their constructor functions. (JavaScript, unlike C++ and Java, allows only one constructor for each class.) After declaring the constructor function, use the DTAPI **`MakeClass`** function to turn the constructor into a class.

For example, to create a `Rectangle` class:

```
function Rectangle(x, y, height, width)
     {
          this.x = x;
          this.y = y;
          this.height = height;
          this.width = width;
     }
MakeClass(Rectangle, PkgRectangle);
Rectangle.InheritFrom(DTObject);
```

The two arguments to MakeClass are the constructor function and the package in which the class is declared. (The `InheritFrom` call is described below under Inheritance.)

After calling `MakeClass(Rectangle, PkgRectangle)`, refer to the class using simply "`Rectangle`." (This is a regular JavaScript function object that has some extra properties and methods, which we refer to as a class.)

Note that in the constructor, you must use the **`this`** keyword to refer to the object being created.

To create a Rectangle object (an instance of the Rectangle class):

```
var r = new Rectangle(2,-2, 10, 50);
```

Object Types

The DTAPI categorizes object classes as object types. In JavaScript, **`typeof`** will always return `"object,"` as JavaScript does not recognize different types of objects created using classes. It is useful to know an object's class in order to determine which methods may be used in relation to it. The DTAPI object framework supplies two functions that will return an object's type (or class): `TypeOf`, and `IsA`.

`TypeOf` takes an object and returns a string containing the name of its class. For example:

```
var s = new Square(10);
alert(TypeOf(s));
```

would display an alert dialog box with `Square` in it. Note that the class name does **not** include the package name.

`IsA` takes an object and a string and returns **`true`** or **`false`** depending on whether the string is the name of the class of the object, or the name of one of its superclasses. For example:

```
var s = new Square(10);
alert(IsA(s, "Square"));
```

would display an alert dialog box with `true` in it.

Inheritance

Inheritance allows programmers to declare a new class which specializes or modifies the behavior of a more generalized "superclass," and is a means by which related, subordinate classes "inherit" methods from their parent class.

In the DTAPI, inheritance is accomplished using the **`InheritFrom`** method.

For example, to create a `Square` class that inherits from `Rectangle`:

```
function Square(x, y, size)
     {
          this.Rectangle(x, y, size, size);
     }
MakeClass(Square, PkgRectangle);
Square.InheritFrom(Rectangle);
```

Note how the `Rectangle` constructor is called within the `Square` constructor. A subclass should always call its superclass' constructor, usually at the beginning.

Note, if a class does not inherit from anything, some things may not work. You should always define a class as inheriting from **`DTObject`**, the base class, if nothing else.

Methods

Methods define and act upon properties of a class object. In JavaScript, methods are also functions.

To define a method, after declaring the method function, use the **`MakeMethod`** method of the class.

For example, to create an `area` method of the Rectangle class:

```
function Rectangle_area()
     {
          return this.height * this.width;
     }
Rectangle.MakeMethod(Rectangle_area);
```

To invoke this method, call:

```
var r = new Rectangle();
alert(r.area());
```

Note that the method function is declared with the name "`Rectangle_area`," but that the method name itself is "`area`." This marks the function as meant to be called as a method on the `Rectangle` class. `MakeMethod` knows that the class name is `Rectangle`, so it strips that part off and leaves only "`area`" as the method name.

Note that in methods you must use the **`this`** keyword to refer to the object on which the method is called.

It is sometimes desirable to change the way a method supplied by the superclass works. For example, the `setSize()` method for the Rectangle class requires both a height and a width parameter. To replace it in the Square class with a method which takes only one `size` parameter, use the `this.Call` method, as shown:

```
function Square_setSize(size)
     {
          this.Call(Rectangle, "setSize", size, size);
     }
Square.MakeMethod(Square_setSize);
```

Note that the `Call` method invokes the method of a superclass on an object, bypassing normal method invocation. Calling `this.setSize(size)` would call the setSize method of the `Square` class, rather than that of its superclass, `Rectangle`.

Persistence

One of the key features of the DTAPI is *persistence*. Persistence allows users to log out of their Desktop, go to a different site or browser, log back in, and still have all of their files, and the appearance of their Desktop exactly as they left them.

To allow users to save information for later retrieval, the DTAPI implements a concept called *Persistent Data*. Persistent data is that which persists from user session to user session, and includes such information as the user's ID and information, and the active or inactive state of the application at the end of the last user session.

The basic unit of persistent data is the *Persistent Object*, or PO, which is an object that is duplicated on the Desktop.com servers. All persistent data is stored in instances of `DTObjectStore.PersistentObject` or one of its subclasses. The properties of POs may be numbers, strings, Booleans, or references to other POs. A group of POs, the next level of abstraction, is called an *Object Group*. An Object Group is a collection of POs arranged in a tree, with a special type of PO at the root of the tree. This special type of PO is the Root Persistent Object, or RPO, which is like other Persistent Objects in that it can have properties, but is special because it is the only type of PO that can be the root of an Object Group.

All Persistent Objects are identified by a three-number tuple. The first number is the user ID, the second is the RPO ID, and the third is the child ID. Each user has their own user ID, and all of that person's POs start with their user ID as the first number in the tuple. The second number identifies an Object Group. Every PO within that group has the same first and second number in the tuple. The third number identifies the particular PO within the group, with the RPO always having a child ID of 0.

Persistent Objects are grouped into bundles under the **`RootPersistentObject`** class. Root Persistent Objects are like files in that they bundle data together. Each `PersistentObject` must belong to a `RootPersistentObject`. (When creating a `PersistentObject`, it must be associated with a `RootPersistentObject`.) When a `RootPersistentObject` is deleted, all of the data associated with it (stored in `PersistentObjects`) is also deleted.

Creating Persistent Objects

Persistent Objects (POs) must be associated with a Root Persistent Object (RPO) when they are created. To associate a PO with an RPO, pass either a `RootPersistentObject` or a `PersistentObject` to the `PersistentObject` constructor. If an RPO is passed, the new PO is associated directly with the given RPO. If a PO is passed, the new PO is associated with the same RPO as the PO that was passed in.

For example, with `rpo` a `RootPersistentObject`:

```
var po_a = new PersistentObject(rpo);
var po_b = new PersistentObject(po_a);
```

creates two `PersistentObject`s that are both associated with the given `RootPersistentObject`.

Working with Persistent Objects

`PersistentObject` properties may contain five types of data:

- strings,
- integers,
- Booleans,
- references to other Persistent Objects, and
- **null.**

The **`set`** method must be used to set properties on a persistent object. For example:

```
po_a.set("myProperty","myValue");
```

would set the `myProperty` property of `po` to "`myValue`," a string. The `set` method must be used to indicate to the object that the value must be sent to the server, and thus made persistent. If `set` is not used, the value will not be saved, and the property may disappear at any time.

The calls to set other data types are similar. For example:

```
po_a.set("myInteger", 10);
po_a.set("myBoolean", true);
po_a.set("myNull", null);
po_a.set("myObject", po_b);
```

To read property values from a `PersistentObject`:

```
var s = po_a.myProperty;
var i = po_a.myInteger;
var b = po_a.myBoolean;
var n = po_a.myNull;
var o = po_a.myObject;
```

Restoring Persistent Objects

To restore a Persistent Object, use the **`fetchChild$`** method. For example, if the `po_a` in the previous example has just been restored:

```
alert(po_a.myObject);        // shows "null"
po_a.fetchChild$("myObject");
var po_b = po_a.myObject;
```

Note that when `PersistentObject`s are restored, references to other Persistent Objects are also restored, but the objects themselves are not.

For more information, and a list of available methods, see DTObjectStore.PersistentObject.

Root Persistent Objects

The `RootPersistentObject` class is a subclass of the `PersistentObject` class. Instances behave exactly like regular `PersistentObject` instances. The only difference is that a `RootPersistentObject` *defines* a group of Persistent Objects; therefore the RPO constructor takes no arguments:

```
var rpo = new RootPersistentObject();
```

As mentioned earlier, a `RootPersistentObject` is like a file: when it is deleted, all data associated with it are also deleted. Also like a file, an RPO is not considered saved unless it is in the **`FileSystem`**. (For more information, see DTFileSystem.) The `RootPersistentObject` type determines what kind of file it is, and what application is associated with it.

The relationship of Persistent Objects and Root Persistent Objects is designed to enhance file management, and limit file proliferation.

Subclassing PersistentObject and RootPersistentObject

A subclass of a `PersistentObject` or `RootPersistentObject` may be generated to create a new file type to associate with an application.

Remember that the constructors for both `PersistentObject` and `RootPersistentObject` take two arguments, even though they are normally called with only one. (The extra argument is used when the object is being restored.) Also remember not to do *anything* in the constructor but call the base class constructor:

```
function MyPersistentObject(arg1, arg2)
     {
          this.PersistentObject(arg1, arg2);
     }
MakeClass(MyPersistentObject, PkgMyApp);
MyPersistentObject.InheritFrom(PersistentObject);
```

If you need to do anything normally done in a constructor, implement the **`construct`** method. As always, call your base class' method first:

```
function MyPersistentObject_construct()
     {
          this.Call(PersistentObject, "construct");
          this.set("myProperty", "myValue");
          this.set("myObject", new
          PersistentObject(this));
     }
MyPersistentObject.MakeMethod(MyPersistentObject_const
ruct);
```

The `construct` method is not a blocking function: no blocking functions may be called within it. To bypass this limitation, make an `initialize$` method to call after creating a new instance.

Note that because constructors are always non-blocking, you may create Persistent Objects in the `construct` method. If you implement a `construct` method, you will probably also have to implement a **`restore$`** and an **`update$`** method.

The `restore$` method is called when an instance is being retrieved from the server and restored, which presents an opportunity to fetch other objects from the server. For example:

```
function MyPersistentObject_restore$()
     {
          this.Call$(PersistentObject, "restore");
          this.fetchChild$("myObject");
     }
MyPersistentObject.MakeMethod(MyPersistentObject_resto
re$);
```

Calling `fetchChild$` gets the associated object from the server. Note that this will delay the calling routine, as it has to wait not only for the current object to be fetched and restored, but also for all of the objects that its `restore$` method fetches.

The `update$` method is similar to the `restore$` method in that it is called when an object is being updated with the latest data from the server. If necessary, you can make `update$` ensure that associated objects are also updated:

```
function MyPersistentObject_update$()
     {
          this.Call$(PersistentObject, "update");
          DTObjectStore.update$(this.myObject);
     }
MyPersistentObject.MakeMethod(MyPersistentObject_updat
e$);
```

Note that each `fetchChild$` and `fetchChildren$` call will make trips to the server if the object is not already located on the client. Using `fetchChildren$` is better than using multiple `fetchChild$` calls, because it groups all of the requests into one server trip. `DTObjectStore.update$` also always causes a server trip. (`Rendezvous` objects may also be used to limit trips to the server.)

The FileSystem

The DTAPI provides a filesystem in which users, applications, and the system itself may store information. The DTAPI filesystem is similar to UNIX or Windows filesystems, but has some important differences.

The primary difference is that the DTAPI filesystem does not store files, it stores objects. Specifically, it stores Persistent Objects and Root Persistent Objects (and their subclasses). When this manual mentions a file in relation to the DTAPI filesystem, it refers to a `RootPersistentObject` and its associated object group.

The second difference is that you do not have to "save" your data. Most traditional programs work on a file that they load into memory. After modifying the data in memory, the program then writes the file back to the filesystem, possibly overwriting the old file. Using the DTAPI, programs work directly with the data in the filesystem; when they make a change, it is immediately saved.

The structure for the DTFileSystem is made up of *nodes*, which are either *directories* or *files*. The relationships among nodes are defined by links: *hard links*, *smart links*, and *symlinks*.

A *hard link* points from a directory node to a node (file or directory) that it contains, and serves as the primary containment relationship between a directory and another node. A node can only have one hard link pointing to it; in this way, every node can be said to exist at a single location in the filesystem.

A *smart link* differs from a hard link in that it is viewed as a secondary link to a node, rather than a link that defines a node's location. Smart links may define secondary routes between directories and/or files in a user's filesystem. Hard links and smart links only reference nodes within a single user's filesystem. Both hard links and smart links are backreferenced: calling `DTFileSystem.getLinks$()` will reveal both the hard links and the smart links that point to a particular node.

A symbolic link, or *symlink,* is a path reference to another node in which only the target node's path is encoded. Symlinks are not backreferenced from their targets, and no validation is done on their target paths until an attempt is made to access those paths. By convention, symlinks are used only to create links to other users' filesystems.

Locations within the filesystem are described as *paths*, which are strings of link names separated by slash characters. Paths may traverse any kind of link: hard links, smart links, or symlinks.

The *root directory* is the top-level directory for a user's filesystem. By convention, the Desktop.com user interface does not expose the root directory to users; instead, the "My Files subdirectory of the root directory is the root of what users may see. Files in other parts of the filesystem are effectively hidden.

Files and Directories

As mentioned above, files in a regular filesystem are `PersistentObject`s in the DTAPI. It is important to keep this in mind because it is often more convenient to talk about "files" than "RootPersistentObjects." For the end-user, there is no perceived difference between a normal file and a DTAPI "file:" users experience `PersistentObject`s as files, manipulate them as files, and call them "files." The distinction between the two is made only for developers.

Like a regular filesystem, the DTAPI filesystem provides a tree-like directory structure in which each directory may contain other directories and files (`PersistentObject`s). Each user has a private directory structure of their own, and it is not possible for one user to access another user's files through the filesystem. (Use the DTFileSharing package to share files or access shared files.)

In a user's directory structure, certain directories contain certain kinds of data. The "`My Files`" directory contains all of the files that the user sees through the file manager. The "`My Desktop`" directory contains everything on the user's Desktop. There are some directories that the system uses. There are some Desktop.com applications that create their own directories under the root (this should generally be avoided). To create a directory for an application, call the **`getAppDirectory$`** method of the `Application` class.

To get at a file you must know its **path**. The path is the list of directories, starting at the root, that need to be traversed to find the file, plus the name of the file, all separated by forward slashes ("/"). To get to the "`Me.gif`" file in the "`Pictures`" directory in the user's "`My Files`" directory, use:

```
/My Files/Pictures/Me.gif
```

The DTFileSystem

The DTAPI package that implements the filesystem is called **`DTFileSystem`**. The `DTFileSystem` package exports many functions, but the two most basic are **`DTFileSystem.get$`** and **`DTFileSystem.put$`**.

`DTFileSystem.get$` retrieves a `RootPersistentObject` from the filesystem using its path. From the `RootPersistentObject`, you may access all `PersistentObject`s in its object group.

For example:

```
var result = DTFileSystem.get$("/My Files/foo");
if (!result.success)
     {
          ...error handling code...
     }

var rpo = result.object;
```

`DTFileSystem.put$` inserts a `RootPersistentObject` into the filesystem at a particular path. `DTFileSystem.put$` is often used after a new `RootPersistentObject` is constructed. If `DTFileSystem.put$` is not used, you will not be able to retrieve your objects.

For example:

```
var rpo = new RootPersistentObject();
var result = DTFileSystem.put$("/My Files/bar", rpo);
if (!result.success)
     {
          ...error handling code...
     }
```

For more information, and a complete list of functions and methods, see DTFileSystem.

GUI Structure

As this API is designed to create end-user applications, the DTAPI offers a broad set of pre-defined GUI enabling packages and concepts. These GUI components are arranged to provide both ease-of-use and expandable functionality to the developer.

Components

A component is an object which defines an element of the graphic user interface for an application. Components may define windows, images, text displays, menus, or any combination of these elements, and may include both interactive elements and layout constraints. Some of the more common components are described here, to serve as examples of those available.

Container components are used to combine other components into a single entity, as experienced by the end-user. The Pane component defines a window in which other components may be placed. A layout manager may be attached to the Pane, which will arrange the elements within it relative to the Pane itself. The Pane and all its contained elements are treated as a single unit by the end-user, and may be moved and resized by simply moving the Pane container. This and its parent class, Container, are both components which may contain other components, including different instances of the same class. This feature introduces hierarchy into the application structure, and is intended to help simply application design.

Text components are used to display and arrange text, and may be interactive. One of the most common of these is TextLabel, which may be used to place text, and set its location and alignment. For example:

```
var textlabel = new DTTextLabel.TextLabel();
textlabel.setLocation(10, 10);
textlabel.setText("Hello World");
textlabel.setAlignment(DTTextLabel.TextLabel.LEFT);
addComponent()
```

This will create a text label of width 50, height 25, font color red, text "Hello World" and text aligned to the left inside the area. The location of this component is set to (10, 10) within the containing container. TextLabel variations include TextImageLabel, which includes an image with the text, and TextBox, which allows multiple lines of text.

Image components display an image. To simply display an image, call an instance of ImageArea, which supports three different modes. The first mode lets you display the image in its natural size, the second lets you stretch or compress the image to fill the area specified by the size of the ImageArea. The third allows you to clip a larger image to the size specified.

Components are meant to be used as building blocks in the creation of an application. As such, they may be combined and re-combined to suit the developers' needs. For example, he Button component combines image and text in a pre-defined format, which includes an action listener. The component will automatically change appearance when clicked, mimicking standard button features. The associated actionListener will emit the appropriate actions when called. This component may be placed in either a window or a pane component.

Layout Managers

Layout Managers are classes whose function is to position the various components in a container. Layout managers are associated with containers, and each has a defined pattern to lay out components added to the corresponding pane. Each time a pane is painted or resized, the `doLayout()` method of the layout manager is called and the components are laid out. Some layout managers also control the size of components contained in the associated panes.

A description of each layout manager is given with its class description. Examples for two common layout managers are given here.

The ArrayLayoutManager stacks all components in an array one after the other in a single row or column. In the example below, the manager is created with the VERTICAL parameter, which will cause it to stack the components along the vertical axis. Different layout managers have different parameters, and are designed for different uses and effects.

To create an ArrayLayoutManager and add it to a container, call:

```
var alm = DTArrayLayoutManager.ArrayLayoutManager;
var lm = new alm(alm.VERTICAL);
pane.setLayoutManager(lm);
pane.addComponent(c1);
pane.addComponent(c2);
     ......
pane.addComponent(cn);
```

Some layout managers take a constraint object, which is a means to define all properties needed to arrange the component. The constraint object is a separate class, an instance of which should be passed as an argument to the addComponent method of the container. For example, the `TableLayoutManager` arranges components in the form of a table, where each component has a cell position associated with it. The cell position, along with the component padding, weight, and spans (number of rows or columns that the component spans) are then specified for each component by setting the properties on an instance of `TableLayoutConstraint`, and passing that as the second argument of `addComponent()`.

To create a Table Layout Manager with component positioning constraints:

```
var lm = new
DTTableLayoutManager.TableLayoutManager();
pane.setLayoutManager(lm);
lm.setRows(2);
lm.setColumns(2);
var constraint =
     new
     DTTableLayoutConstraints.TableLayoutConstraints()
     ;

constraint.setRowNumber(0);
constraint.setColumnNumber(0);
constraint.setColumnSpan(2);
constraint.setRowSpan(1);
pane.addComponent(c1, constraints);
constraint.setDefaults();
constraint.setColumnNumber(0);
constraint.setRowNumber(1);
pane.addComponent(c2, constraints);
constraint.setDefaults();
constraint.setColumnNumber(1);
constraint.setRowNumber(1);
pane.addComponent(c3, constraints);
```

This will layout the components as shown below.

c1

c2 c3

Common windows and dialogs

Window components define the area in which objects appear, and may include Application Windows, Dialog Windows, Alerts, and input boxes. Window components may be associated with layout managers and action listeners to define desired GUI elements.

Two of the most commonly used windows are instances of `AppWindow` and `DialogWindow`. AppWindows are persistent: unless the window is closed, it persists from one Desktop session to the next. They have a title bar and may be resized.

DialogWindows are similar to AppWindows in that they are always associated with an application, but differ in dialog windows' specificity. DialogWindows are normally associated with a window belonging to an application. DialogWindows are not persistent: they do not persist from one session to another. They are not resizable by the user. (The application and other widgets may manipulate a dialog's size by calling the relevant methods.) DialogWindows maintain focus. When a DialogWindow is launched, it stays in front of the other window and retains the focus until it is closed. The other window cannot be raised to the forefront as long as the DialogWindow is open.

AppWindows and DialogWindows are created and used similarly. To create a dialog window:

```
var dialog = new DTDialogWindow.DialogWindow();
dialog.setLocation(100, 100);
dialog.setSize(200, 250);
dialog.setTitle("This is a dialog");
app.addDialog(dialog, "dialog_reference");
dialog.setLayoutManager(layout_manager);
     .....
dialog.addComponent(comp);
     .....
```

To make commonly used dialogs, such as Alert and Question, easier to use, the DTAPI provides a wrapper function, which allows these dialogs to be called directly. For instance:

```
DTAlert.DoAlert$("this is the alert text", app, win);
```

or

```
DTAlert.DoAlert("this is the alert text", app, win,
cb)
```

The first argument is a string representing the text to be displayed in the alert box. The second argument (optional) is a reference to the Application associated with the alert. The third argument (optional) is a reference to the window in which the Alert should be centered. If the first (blocking call) syntax is used, then this call blocks until the user hits the "OK" button on the alert box. If the second syntax is used and the cb is an instance of Callback, then the Callback gets called whenever the user hits "OK." If `cb` is `null`, this is a non-blocking call, and the program continues without waiting for user input.

To create a Question dialog, call:

```
DTQuestion.AskQuestion$
     ("this is the question", app, win);
```

or

```
DTQuestion.AskQuestion
     ("this is the question", app, win, cb);
```

Note that the syntax to create both these dialogs is the same; only the return values will differ. Alert will return only OK, while Question will return Yes, No, or Cancel.

Event Handlers and Action Listeners

Listening

Some Components must be able to react to events received in others. For example, a menu bar, which is a composite Component, must respond to mouse clicks in the text labels that anchor its pull-down menus. This kind of event distribution can be achieved with ***event listening***. Any Component can register as an event listener with any other Component. When a Component receives an event, it passes the event on to all of its event listeners, even when the event has local significance for the Component receiving it.

It is sometimes more convenient to listen for ***actions*** rather than for events. Actions are higher-level occurrences than events, and are generated by Components rather than by input devices. Examples of actions include button presses, text value changes, and moves and resizes.

Actions

Actions are generated when a user clicks (`click`) or double-clicks (`dblclick`) on a component, or can be created by a component in response to defined keyboard or timed actions. Any object (any component, application, window, or other object) may listen for actions on any component. (To identify the component, the developer may also set a string as a label for a component using the `setActionLabel` method.) Each component can have only one label, and this label may be overwritten at any time.

A developer can attach any object as an action-listener to a component by calling `addActionListener` on the component. The first argument to `addActionListener` is a reference to the object that will act as a listener. The second argument (optional) is the name of a method (as a string) defined in the listener that is called when a defined action is performed on the component. (If the method name is not provided, it is set to `actionPerformed` by default.) Each time an action is emitted by a component, this method is called with three arguments:

> `object`: a reference to the component on which the action was performed.
>
> `label`: the label of the component; `null` if the label is not set.
>
> `action`: the name of the action performed on the object (component) generating the event.

In addition to the actions provided by the system, a component class may define its own actions, which may be used to communicate information to various listeners. For more information, see the following methods in DTComponent:

```
setActionLabel(label)
addActionListener(object, method)
```

Events

Events are lower-level occurrences than actions, in that they are generated by a simple mouse or keyboard event, rather than by a component.

To listen for keyboard events from a component, and respond to them, implement `handleKeyboardEvent()`. This method is called each time a user types a key on the keyboard when the focus is on the component. (The concept of focus is explained later in this chapter.) For more information, see `handleKeyboardEvent()` in DTComponent.

One can also listen for mouse actions (like mouseup, mousedown, etc.) by adding a listener to a component by calling `addMouseListener()`. This, like `addActionListener()`, takes an object (listener) as the first reference, and a name of a method in the listener (as a string) as an optional second argument. (The default for the second argument is `handleMouseEvent`.) Methods of listeners are called each time the mouse interacts with the component, and take only one argument: `event`. The event argument contains all the information and reference to the object. For more information, see the following methods in DTComponent.

```
addMouseListener(object, method)
handleMouseEvent(event)
```

Both `handleKeyboardEvent()` and `handleMouseEvent()` are propagated up the container hierarchy. If a component does not respond to an event, then the container containing the component will receive the event. This event propagates up until a container responds to it, or the top-level container is reached.

Focus

In the DTAPI's mouse-and-keyboard interface, the purpose of focus is to direct keyboard input to the appropriate Component. Focus is a means of defining the primary, active component on the screen.

In the Desktop environment, any component can *get* focus, either when the user clicks on the component, or because a widget or an application sets the focus on the component by calling `setFocus(component)` on the containing pane. Once in focus, the component will respond accordingly: its appearance may change if alternate images have been defined for active and inactive states, and all keyboard events will be routed to it.

Unless otherwise set, every component is focusable by default. To deny a component focus, call `setFocusable(false)`.

Panes

The Pane class is a subclass of Container that has the ability to manage its children (all Components) with a concept of ***focus***. At any time, zero or one of a Pane's focusable children (a class-level Component property) may have the Pane's focus. If there is a child in focus, it is only ***active*** if the Pane itself has focus within its parent Pane, and so on all the way up to the Screen, which is the top-level Pane. In tracing down the Pane/Component containment tree from its root at the Screen, there is only one path along which Components are active: the ***active path***. The deepest Component in the active path is called the ***primary*** Component, and may be a Pane, or a non-Pane Component (such as a dialog window). There is always exactly one primary Component on the Screen. Components (including Panes) always know whether they are active, and the WindowManager (which manages the Screen) always knows which Component is primary.

A Pane maintains a stable ordering of its focusable children, called the ***focus ring***. This ordering is used to define a serial path of user movement among Components. A Pane also maintains a flag that indicates what to do when the end of the focus ring is reached: cycle around to the beginning, or ignore the focus advance and allow the parent to handle it. (This is useful for things such as tab windows and radio button groups.) Focus rings may consist of an arbitrary interleaving of widgets and nested Panes.

A Pane may have one of its child Components designated as its ***default focus***. The default focus may be ***strong*** or ***weak***. A strong default focus (which may be null) always attempts to take focus when a Pane becomes active (a previously focused Component may refuse to allow this). A weak default focus takes focus when a Pane becomes active and has no previously focused Component.

Inactive Panes remember focus. When a Pane becomes active again, it can reestablish the active path that existed below it when it was last active. This is useful, for example, in maintaining independent focus states for separate application windows. The focus remembered by inactive Panes does not affect event routing.

Containers

It is possible to use Containers that are not Panes. Typically, non-Pane Containers are used to combine multiple Components into a single widget. For example, a scrollbar is a Container containing three buttons, one of which moves, while the other two are fixed at the ends of the scrollbar.

In handling focus-related events, containers are treated differently than Panes. Non-Pane containers and their children act as a single component with regard to refocus events, and children of non-Pane Containers are never considered active. Components placed in a container will not receive any actions that deal with focus, and will, instead, pass such actions up the focus path. If a user clicks on a component inside a container, the keyboard events will go directly to the container, whereas if the component were contained in a pane, the keyboard events would go to the individual component.

Focus Events

All ***keyboard events*** are routed to the primary Component, which may decide either to handle the event, or pass it up to its parent Pane.

All ***mouse events*** are routed to the Component from which they originate. That Component may decide to treat the event as a ***refocus event***, which causes the Component to become primary if it is not already active. (While Components may define any mouse event type or types to be refocus events, the default is to treat only mousedown events as such.) Any mouse event not treated as a refocus event is a ***pass-through event***, handled normally by the Component.

Keyboard Events

When a Component becomes primary, it registers with the WindowManager to receive all keyboard events. When a keyboard event occurs, it is routed to the primary Component.

When the primary Component receives a keyboard event, it examines the key combination in order to decide what to do. If the key combination has local significance for the Component, it handles the event itself; if not, the event is passed up to the Component's parent Pane, which then examines the key combination in the same way. The event continues up the active path until a Pane is reached which ascribes significance to the event, and decides to handle it. If no Pane along the active path (including the Screen) is interested in the key combination, the event is discarded.

Note that if multiple Components along the active path attach meaning to the same key combination, the most deeply nested Component among these, that equal to or closest to the active component in the chain, will handle events having that key combination. For example, a dialog box may assign the same meaning to the Enter key as pressing the OK button. If that dialog box contains a multi-line text input field, the more local meaning, "new line," will be attached to the Enter key when the focus is in the text input field.

Focus-Advance Events

Focus-advance events use the focus ring, allowing users to advance in order through the Components in a Pane, without using the mouse. For almost all situations, a Pane subclass should interpret a pair of related key combinations (such as Tab and Shift+Tab) as focus-advance keys, one for the forward, and one for the reverse direction.

A Pane can operate its focus ring in one of two modes: circular or non-circular. In circular mode, advancing in either direction beyond the end of the focus ring loops the focus back around to the Component at the other end. In non-circular mode, attempting to advance beyond the end of the focus ring causes a Pane to ignore the focus-advance event, and pass it up to its parent Pane to handle. This has the effect of "jumping" focus out of a Pane to the Pane’s parent. This can be useful, for example, in implementing a control-group container (such as a radio-button group) as a Pane. The controls in the group will appear to the user to be part of the parent Pane’s focus ring.

Refocus Events

When an inactive Component receives a mouse event, it examines the event type and decides whether the event is a refocus event. The decision is made according to a dynamic Component property that indicates what event type or types, if any, are to be treated as refocus events for that Component. The default for this property defines only the mousedown event as a refocus event.

If a received event is not a refocus event, the receiving Component ignores the event. If the event is a refocus event, the Component passes the event to its parent Container. That Container may or may not be a Pane.

If a Pane receiving a refocus event from a child Component is not active, it routes the event up to its parent Container, and so on up to the Screen, which is by definition always active. Eventually, through this process, an active Pane will be found; called the ***pivot Pane***. The pivot Pane is the deepest common ancestor of both the old and new active Components in the Component containment tree.

Once the pivot Pane has been found, it asks its focused Component to lose focus. That Component may perform an internal consistency check to ensure that no user-supplied data is invalid, refusing to give up the focus if this is the case. If the Component being asked to lose focus is a Pane, it skips the consistency check and informs its own focused Component that it will be losing active status (although keeping focus). The no-longer-active message is propagated down along the active path to the primary Component.

If and when the pivot Pane successfully gets its previously focused Component to lose focus, it then grants focus to the child Component that routed the refocus event up to it, making that Component active. If the newly focused Component is a widget, it becomes primary. If it is a Pane, it repeats the same logic as the pivot Pane with respect to taking focus away from its currently focused child Component. Again, a focused Component in a newly activated Pane has the opportunity to refuse to relinquish focus, in which case it remains active, and if it is itself a Pane, the active status propagates all the way down the focus chain below it. If, on the other hand, the currently focused child of the newly active Pane agrees to give up focus, then focus is granted to the child Component that routed the refocus event up to that Pane, making the child active. The propagation of active status then repeats down the chain until either the Component where the refocus event originated becomes active (and primary), or the activation chain is sidetracked somewhere along the way by a refusal to give up focus.

If and when the originating Component becomes primary, it also handles the event that caused the refocus to occur. For example, if a mousedown event causes a focus change, mousedown is delivered to the Component where the event occurred once that Component has become primary. If the Component that becomes primary is a non-Pane Container, and the event originated in one of its child Components, the Container passes the event back down to the originating child to handle. If activation is sidetracked somewhere by a refusal to give up focus, the refocus event is discarded. This ensures that mouse events are only delivered to the Components where they occur.

In summary, a refocus event causes an *up-down-down* chain of messages. The event moves up the containment tree from the originating Component until it reaches the pivot Pane. Deactivation is then propagated down from the pivot Pane along the active path to the then-primary Component. Activation and focus granting is then propagated down from the pivot Pane to the originator.

Programmatic Focus Changes

It is possible to set the focus within any Pane programmatically. If the Pane for which the focus is being set has a focused Component, that Component may refuse to give up focus. This refusal can be detected by the code attempting to set the focus. If the Pane for which the focus is being set is active, the previously focused Component, if any, is encouraged to display a warning dialog if it is refusing to give up focus.

Application Modes

Launching Applications

There are several ways to launch an application. Applications may be launched directly by double-clicking an application icon on the Desktop, or by selecting the application from the "My Apps" menu. Both of these methods call

```
DTApplicationManager.OpenApp(appname)
```

where the `appname` is the name of the application's Package. Applications may also be launched indirectly by launching a file which is associated with an application.

Launching a File

Applications can be launched indirectly by double-clicking a file in the File Manager, which launches the application associated with the file in the system's Type Registry. The Type Registry maintains a mapping between various file types and the applications that are capable of launching them. When launching a file the function

```
DTApplicationManager.OpenFile(filepath)
```

is called, with `filepath` the path to the file to open.

Single Instance

By default, all applications enforce a single instance rule. If a user tries to relaunch an open application, the application window of the first instance will be displayed instead of relaunching the same application. This default allows the application to determine how to handle multiple application windows and files.

If the application is directly launched and an instance already exists, then `DTApplicationManager.OpenApp()` will call its `callInstanceHandler(signal)` method with `signal` defined as `openapp_single_instance`.

If a file is being launched and an instance already exists, then `DTApplicationManager.OpenFile()` will call its `callInstanceHandler(signal, rpo)`, where `signal` is `openfile_single_instance`, and `rpo` is a reference to the file object.

Multiple Instances

Multiple instances of an application may be used to simplify applications. A single instance typically can display only one file at a time. By enabling multiple instances, the application logic is simpler than supporting multiple files in a single application instance.

To override the default Single Instance behavior for an application, the application's constructor needs to define a property named `SINGLE_INSTANCE` to be false. For example:

```
function DeskPadBase()
     {
          this.Application();
          this.SINGLE_INSTANCE = false;
     }
MakeClass(DeskPadBase, DTDeskPadBase);
DeskPadBase.InheritFrom(DTApplication.Application);
```

DeskPad inherits from a class called DeskPadBase. Its constructor function sets the `SINGLE_INSTANCE` property to `false` to indicate that more than one instance of the application should be allowed.

Application Data Types

There are five general types of data that exist in an application: *application object properties*, the *state object*, the *preference object*, *files* in the filesystem, and the *hints object*. Data that is put in an application should go into one of the five general types.

Application data is divided into these types to allow the system to optimize data management in an application framework. Different types of data have different lifetimes in an application. The application framework manages when application data is no longer needed, and does garbage collection. Assigning data to different data types helps structure the design of an application.

The *application object* stores the most volatile application data properties, and is where transient data should be stored, such as the current width and height of an application window. Application properties may be set and changed using the `this` pointer in the application.

The *state object* is designed to store persistent data across user sessions. This object stores data that may be retrieved when an application is reopened after having been closed. For example, a count may be made each time an application is started and kept in the state object, to determine the number of times the application has been executed. This object is accessed through the `DTApplication.getStateObject()` method.

The *preference object* stores persistent data that concerns the configuration of an application. For instance, a telnet client application would store the last host and port number to which the client connected in the preference object. This object may be accessed through the `DTApplication.getPrefs$()` method.

A *file* in a user's filesystem may be used to store data required by the user to interact with other applications or users. This data is application specific, and may be a text file, a GIF picture, or any other appropriate file type. To access files, use the DTFileSystem API. (See DTFileSystem for details.)

Hints are the final type of data used in an application. Hints are tied to files, and are meant to tell how an application should interact with a file. A file's payload type might be stored in a hints file, indicating whether it is a binary or text file. If binary, it can specify a mime type for the file, which the application will use to determine what to do with the file. There are no standard formats for a hints object. To access the hints object, use the `DTApplication.getHints$(rpo)`, where `rpo` belongs to the file in question.

Data Name	Access	Lifetime	Scope
App Object	this.foo	Session	Instance
State Object	DTApplication.getStateObject()	Session	Instance
Prefs Object	DTApplication.getPrefs$()	Account Life	User+app
Files	DTFileSystem API	User Defined	User+filesystem
Hints	DTApplication.getHints$()	User Defined	User+app+file

Definitions:

Session: the time that the user is logged on

Account Life: the time that the user account exists in the system

User Defined: determined by the user's actions

Instance: a running application

Writing an Application

This chapter describes Devtool, Desktop.com's developer tool, and outlines the tools available and steps necessary to develop an application for Desktop.com.

Submitting Apps to Desktop

To submit an application for inclusion on the Desktop.com website, please send an email describing the application to: devapps@desktop.com.

Devtool

Devtool is Desktop.com's developer tool. It provides a unified interface for functions needed to edit and test an application online. Devtool provides the ability to:

- upload and download code to your account
- compile JavaScript for execution under the Desktop environment
- execute application code
- assemble application projects consisting of multiple files
- "touch-up" edit code on-line
- view debugging output from apps
- access the console, Desktop's command-line interface

Devtool may be accessed from the **My Apps | Dev Tools | Devtool** pull-down menu item on your Devtop account.

Devtool's interface consists of an area for application debugging output, an area for output from the code processor, a menubar, and a toolbar that provides access to its most commonly used functions.

Devtool also includes a simple text editor, Edwin, that may be launched by creating a new file or opening an old file. Multiple editors may be open at any time.

All open files that are valid Desktop packages become part of your project. Typically, your projects will consist of only a few packages at most, one of which will be your application package. The others are likely to be subclasses built in conjunction with the application. Save your project from the **Project | Save** pull-down menu inside Devtool, or by clicking on the save-project button on the toolbar.

Editing Code

While developers need a full-featured text editor to write applications and make major changes, Devtool provides the means to make quick fix modifications to code on-line. The ability to edit on-line allows developers to debug without uploading and downloading each time a fix is necessary. We suggest that you edit code locally, then upload it to your Devtop account when you're ready to try it out.

Images and HTML

To incorporate images and HTML in your applications, store files in your filesystem, and reference them using the path prefix, "/cgi-bin/getfile/." For example, for an image called "frog.gif" in the directory named "/My Files/images," reference the image using

```
/cgi-bin/getfile/My%20Files/images/frog.gif
```

(Note that it is necessary to escape any whitespace characters with "%20.")

Uploading and Downloading Code

To upload code, login and launch Devtop, then select **File | Upload**, or click on the file-upload button in the toolbar. The file-upload browser will appear, prompting you for the path from which to fetch the code locally, and to which to store the code on your Devtop account. The uploaded file will automatically open in a new editor window (Edwin) after it has been uploaded and saved.

You may also upload files using the Desktop **My Files** pull-down menu, or File Manager, without launching Devtool. These files may be loaded into Devtool by selecting **File | Open** from the pull-down menu in Devtool, or by clicking on the open-file button on the toolbar.

(Note that a project may consist of files placed in multiple directories, allowing you to organize files however you like.)

If you make changes to your application online, you may want to download the modified code to keep your local copy up-to-date. To download files individually from Edwin, select **File | Download**, or click on the file-download button on the toolbar. You may also download files using the Desktop **File** pull-down menu or File Manager, without launching Devtool.

Compiling

Devtool takes your application code, parses it for errors, and formats it for the Desktop environment. It reports syntax errors, undeclared symbols, and unused symbols. The changes it makes include stripping whitespace and comments, translating blocking methods, and renaming identifiers.

When your application code is loaded in Edwin and ready to be compiled, select **Tools | Compile** from Devtool's pull-down menu, or click on the **Compile** toolbar button. Devtool will begin processing your code, and the results will appear in the left-hand or upper section of the Devtool interface. If successful, Devtool will write out a compiled version of your code as a ".ojs" file: the file that is loaded when you run an application.

(Selecting **Tools | Compile and Run** will compile the project, then launch the application in the Devtop window.)

Debugging

Devtool provides a simple debugging interface. The secondary window of the application (left-hand or bottom section of the interface) supports text output from your applications.

To generate output from an application in the Devtool debugger, call your application's built-in `debug()` method and pass it the text you would like to see as output. For example,

```
this.debug("hello world!");
```

will send the text "hello world!" to Devtool's debug window.

We recommend that you use Microsoft's Visual Studio and Netscape's Debugger in conjunction with Devtool. These tools provide watch, breakpoint, immediate execution, and other useful features. For more information, see http://msdn.microsoft.com/vstudio/ and http://developer.netscape.com:80/docs/manuals/jsdebug/index.htm, respectively.

The Console

The Console is Devtop's command-line interface, which allows you to manipulate files in your filesystem, run applications, execute JavaScript commands, and more. To launch the Console, select **My Apps | Dev Tools | Console** from the Devtop menu, or **New Console** from the **View** menu in Devtool. Type "help" at the command line to get started.

API Reference Manual

This Desktop API Reference Manual provides descriptions and definitions for all DTAPI packages, classes, methods and functions available to developers.

The DTAPI is designed to be used in conjunction with the Developers' Guide. While the DTAPI serves as a reference manual, the Developers' Guide covers overriding principles and procedures used in implementing the DTAPI. Both manuals often make reference to related information in the other.

Desktop.com Packages

Each core Desktop.com package has a name that begins with the prefix "DT." Each package defines an interface consisting of *classes* that may be instantiated, and *functions* that may be called, which are accessed as properties of the package. For example:

```
var button = new DTButton.Button();      (class instantiation)
button.setText(username);                (method call)
var username = DTUserData.getLogin();    (function call)
```

Each class in a package defines a set of *methods* that may be called on objects of that class. As in most object-oriented environments, classes inherit the methods defined by their superclasses, except where they override such methods. Each class has a special method called the *constructor* that serves to instantiate the class. The class instantiation line in the example above shows a call to the `Button()` constructor, which instantiates and returns a new Button object.

There are a few classes and functions that are *global symbols*, meaning that they may be referenced without the package qualifier. For instance, all of the functions in DTObjectFramework are global symbols, therefore the call

```
if (IsA(object, "Button"))
```

is exactly equivalent to

```
if (DTObjectFramework.IsA(object, "Button"))
```

In order to write a Desktop.com application, you must implement a package of your own. To define and implement an application package, see the chapter "Developer Defined Packages."

Class Hierarchy

The DTAPI classes are arranged in a hierarchical structure, which is shown in the chart below. Their descriptions are arranged alphabetically in this manual, for easy reference.

```
DTObjectFramework.DTObject
 |
 + DTApplication.Application
 |
 + DTCallback.Callback
 |   + DTNamedCallback.NamedCallback
 |
 + DTContent.Content
 |   + DTComponent.Component
 |       + DTContainer.Container
 |       |   + DTBasicButton.BasicButton
 |       |   |   + DTButton.Button
 |       |   + DTCheckBox.CheckBox
 |       |   + DTMenu.Menu
 |       |   + DTMenuBar.MenuBar
 |       |   + DTPane.Pane
 |       |   |   + DTBorderedRadioButtonGroup.BorderedRadioButtonGroup
 |       |   |   + DTDropDownComboBox.DropDownComboBox
 |       |   |   + DTHTMLBrowser.HTMLBrowser
 |       |   |   + DTLabelledTextBox.LabelledTextBox
 |       |   |   + DTRadioButtonGroup.RadioButtonGroup
 |       |   |   + DTScrollPane.ScrollPane
 |       |   |   |   + DTIconsView.IconsView
 |       |   |   |   + DTListBox.ListBox
 |       |   |   |   + DTScrollingTextBox.ScrollingTextBox
 |       |   |   |   + DTTextTreeView.TextTreeView
 |       |   |   + DTTabView.TabView
 |       |   |   + DTWindow.Window
 |       |   |       + DTBorderedWindow.BorderedWindow
 |       |   |           + DTAppWindow.AppWindow
 |       |   |           + DTDialogWindow.DialogWindow
 |       |   |               + DTReusableDialog.ReusableDialog (private)
 |       |   |               |   + DTAlert.Alert
 |       |   |               + DTTransientDialogWindow.TransientDialogWindow
 |       |   |                   + DTSlideDialog.SlideDialog
 |       |   + DTProgressBar.ProgressBar
 |       |   + DTRectangle.Rectangle
 |       |   + DTScrollBar.ScrollBar
 |       |   + DTSlider.Slider
 |       |       + DTIncrSlider.IncrSlider
```

(continued from

```
 |        +  DTContainer.Container)
 |
 |        +  DTDetailsRow.DetailsRow
 |        +  DTEventGrabber.EventGrabber
 |        +  DTHTMLBox.HTMLBox
 |        +  DTImageArea.ImageArea
 |        |    +  DTImageButton.ImageButton (on Internet Explorer)
 |        +  DTImageButton.ImageButton (on Netscape Navigator)
 |        +  DTNativeComponent.NativeComponent
 |        |    +  DTNativeTextInputBox.NativeTextInputBox
 |        +  DTTextBox.TextBox
 |        |    +  DTLinkArea.LinkArea
 |        |    +  DTTextImageLabel.TextImageLabel
 |        |    +  DTTextLabel.TextLabel
 |        +  DTTextInputBox.TextInputBox
 |        +  DTTextInputBox2.TextInputBox2
 |             +  DTWrappingTextBox.WrappingTextBox
 |
 +  DTFileUpload.FileUpload
 |
 +  DTFontProber.FontMetrics
 |
 +  DTLayoutManager.LayoutManager
 |    +  DTArrayLayoutManager.ArrayLayoutManager
 |    +  DTAttachmentLayoutManager.AttachmentLayoutManager
 |    +  DTColumnLayoutManager.ColumnLayoutManager
 |    +  DTFlowLayoutManager.FlowLayoutManager
 |    +  DTGridLayoutManager.GridLayoutManager
 |    +  DTTableLayoutManager.TableLayoutManager
 |
 +  DTObjectStore.PersistentObject
 |    +  DTApplicationStateObject.ApplicationStateObject
 |    +  DTFileSharing.AccessControlList
 |    +  DTObjectStore.RootPersistentObject
 |    |    +  DTApplicationPrefsObject.ApplicationPrefsObject
 |    |    +  DTBookmarks.BookmarkRecord
 |    +  DTPersistentArray.PersistentArray
 |
 +  DTQueue.Queue
 +  DTRendezvous.Rendezvous
 +  DTStyle.Style
 +  DTTimer.Timer
```

DTObjectFramework

The DTObjectFramework package defines the Desktop.com object framework, an extension of the object-oriented capabilities of JavaScript.

Classes

DTObject

DTObject is the root base class for all DTAPI classes. It has no special properties or methods.

All developer-defined classes should be defined as inheriting from (at least) DTObject.

Functions

The DTObjectFramework package defines three kinds of functions. The first kind are ordinary functions, and are all global symbols. These include `MakeClass()`, `TypeOf()`, and `IsA()`.

The second kind are invoked as methods of classes (technically, as methods of class constructor functions). This is in contrast to ordinary methods, which are invoked on objects rather than on classes. These include `MakeMethod()`, `SetDefault()`, `MakeConstant()`, and `InheritFrom()`.

The third kind are invoked as methods of packages (technically, as methods of package functions). `Export()` is the only function of this kind.

`Export(f, function_name)`

Makes the function `f` visible as a property of the package on which `Export()` is being invoked. `f` is specified by reference, and may be either an ordinary function or a class constructor function. It is never necessary to export class methods, since they are invoked through objects rather than packages.

A package should only export those classes and functions that are part of the package's public API. Any classes and functions used only internally within the package should not be exported.

If a string is supplied for `function_name`, it will be used as the name of the package property that references `f`. Otherwise, the name of the property will be taken from the name of `f` (the usual case).

`InheritFrom(superclass)`

Establishes the superclass of the class on which `InheritFrom()` is being invoked. `superclass` is a reference to the constructor function of the desired superclass.

`IsA(o, class_name)`

Returns `true` if `o` is an instance of the class whose name is `class_name`; `false` if not. `class_name` may name a DTAPI class or a built-in JavaScript class. Note that instances of a subclass of `class_name` are also considered instances of `class_name`.

`MakeClass(f, pkg)`

Makes the constructor function `f` into a DTAPI class within the package `pkg`. Both `f` and `pkg` are specified by reference.

`MakeConstant(property, value)`

Establishes a *class property* with the name `property` and the given `value` for the class on which `MakeConstant()` is being invoked. A class property may be accessed later with the syntax *classname.property*. Note that a class property differs from a default property in that it is never visible as a property of instances of the class, only as a property of the class itself.

`MakeMethod(f, method_name)`

Makes `f`, a function specified by reference, into a method of the class on which `MakeMethod()` is being invoked.

If a string is supplied for `method_name`, it will be used as the name of the method. If not, the name of the method will be taken from the name of `f` (the usual case). If `method_name` is not supplied and `f` has a name of the form *classname_methodname*, where *classname* matches the name of the class on which `MakeMethod()` is being invoked, the method's name will be *methodname* rather than the full name of `f`.

`SetDefault(property, value)`

Establishes a *default property* with the name `property` and the given `value` for the class on which `SetDefault()` is being invoked. This property will become a property of every instance of the class. The value of `property` may be overridden in any instance by assigning a different value to it. Overriding the value of `property` for an instance will not affect the default value.

`value` may be of any type. If `value` is a reference type (references to objects, arrays, functions, etc.), the value of `property` will refer to the same object in all instances of the class. If the object is modified, it will appear modified for all instances of the class.

`TypeOf(o)`

Returns a string that indicates the type of `o`. If `o` is an instance of any DTAPI class or built-in JavaScript class, `TypeOf()` returns its class name. Otherwise, `TypeOf()` returns the basic type of `o`, as per the `typeof` operator.

DTAlert

The DTAlert package defines the `DoAlert$()` function

Classes

None.

Functions

`DoAlert$(text, where)`

Displays an alert dialog window with the specified `text` and an OK button. `where` (optional) specifies a reference to an object (either an Application or a Window) relative to which the dialog is to be centered and made modal. If `where` is omitted, the dialog will be centered on the screen and made system-modal. As window centering and modality is the most common, pass a Window reference for `where` unless there is some reason not to.

If `DoAlert$()` is called as a blocking function, it will not return until the user clicks the OK button in the dialog that is displayed.

`DoAlert$()` is preferable to the JavaScript `alert()` because it is implemented within the Desktop.com environment, which allows more control over appearance and provides the user with feedback that clearly comes from within Desktop.com.

Dialog windows are resized to fit the text. `text` is wrapped where appropriate, may include HTML tags, and will translate "`\n`" appearing in `text` as a line break.

DTApplication

The DTApplication package defines the Application class.

Classes

Application

The abstract base class for all applications. To implement a Desktop.com application, create a subclass of the Application class.

Inherits from DTObjectFramework.DTObject.

Constructor

None. The ApplicationManager automatically instantiates Application objects: they should never be instantiated directly.

Service methods

(These methods may be called as properties of `this` by subclasses.)

addDialog(dialog, tag)
addWindow(win, tag)
debug(output)
exit()
getAppDirectory$()
getAppArguments()
getDialog(tag)
getDialogs()
getPrefs$()
getStateObject()
getWindow(tag)
getWindows()
removeDialog(tag)
removeWindow(arg)
setStateObject()

Overridable methods

(These methods may be implemented by subclasses.)

begin$()
getDefaultPrefsObject()
go$()
handleWindowAction(object, label, action)
handleWindowClose$(win)
restoreAfter$()
restoreBefore$()
restoreDuring$()
startup$(rpo)

`addDialog(dialog, tag)`

Draws a dialog object and adds it to the list of dialogs associated with the application instance. `dialog` is a reference to a DialogWindow object that has been appropriately initialized. `tag` (optional) is a string, to be used as a key to refer to the dialog being added. If omitted, a tag unique within the application instance will be automatically generated and given as the return value from `addDialog()`. (`getTag()` may be called on any DialogWindow object.)

All dialogs displayed using `addDialog()` are *application-modal*, which means they must be dismissed before the user may interact with the application displaying them.

`addWindow(win, tag)`

Draws a Window object and adds it to the list of windows associated with the application instance. `win` is a reference to a Window object (usually a DTAppWindow.AppWindow) that has been appropriately initialized. `tag` (optional) is a string to be used as a key to refer to the window being added. If omitted, a tag unique within the application instance will be automatically generated and given as the return value from `addWindow()`. (`getTag()` may be called on any Window object.)

Calling `addWindow()` causes both steps of window initialization, chrome and interior drawing, to occur, and may be called at any time. When `addWindow()` is called from within the `startup$()` method, chrome drawing occurs immediately; interior drawing occurs after `startup$()` returns. If `addWindow()` is called from somewhere other than `startup$()`, and both window drawing steps should be completed before continuing, call `addWindow$(win, tag)` (note the blocking syntax).

`begin$()`

Called as the first hook method in both the startup and restore initialization sequences. This method provides an easy way to write code common to both sequences.

`debug(output)`

Checks for the existence of Devtool, and passes along the application reference. `output` is a string which will be displayed in Devtool's debug window.

`exit()`

Unconditionally destroys the application instance and all its windows.

`getAppArguments()`

Returns the startup arguments, if any. (See the DTApplicationManager functions: OpenAppWithArguments, and OpenFileWithArguments, for more information.)

`getAppDirectory$()`

Finds or creates a directory unique to this application class and returns its path. This directory is available only to the application class, and provides a location to store application-specific data that should not be visible to users. The return value from `getAppDirectory$()` is an object with properties `success`, `error`, and `path`. If `success` is `true`, then `path` is the path to the directory. If `success` is `false`, then `error` is an error code that may be compared against the values in `DTApplication.ERRNO`, or turned into an error message using `DTApplication.getErrorString()`.

`getDefaultPrefsObject()`

Provides a way to specify default preferences for the application class. If implemented, this method is called when `getPrefs$()` is called and cannot find an existing ApplicationPrefsObject in the filesystem. Should return an instance of an ApplicationPrefsObject (or a subclass), or `null` to indicate failure.

`getDialog(tag)`

Returns a reference to the DialogWindow object associated with `tag`; `null` if no dialog with the given tag exists.

`getDialogs()`

Returns an array of all DialogWindow objects associated with the application instance.

`getPrefs$()`

Obtains and returns a reference to the ApplicationPrefsObject associated with the application class. To obtain this object, `getPrefs$()` first checks the filesystem, and returns an existing ApplicationPrefsObject, if available. Next, it looks in the application class, and returns the object returned by the `getDefaultPrefsObject()` method, if that method is implemented. If neither search finds an ApplicationPrefsObject, `getPrefs$()` returns an empty ApplicationPrefsObject.

If a new ApplicationPrefsObject is created, either by `getDefaultPrefsObject()` or by default, it is put into the filesystem and returned by the next call to `getPrefs$()`.

Returns an object with properties `success`, `error`, `existing`, and `object`. If `success` is `true`, `object` is a reference to the ApplicationPrefsObject, and `existing` is `true` if the `object` was found in the filesystem, and `false` if the `object` was created. If `success` is `false`, `error` is an error code that may be compared against the values in `DTApplication.ERRNO` or turned into an error message using `DTApplication.getErrorString()`.

`getStateObject()`

Returns a reference to the ApplicationStateObject associated with the application instance. If called and no ApplicationStateObject exists for the application instance, an empty ApplicationStateObject is created and returned.

`getWindow(tag)`

Returns a reference to the Window object associated with `tag`; `null` if no window with the given tag exists.

`getWindows()`

Returns an array of all Window objects associated with the application instance.

`go$()`

Called as the last hook method in both the startup and restore initialization sequences. This method provides an easy way to write code common to both sequences.

`handleWindowAction(object, label, action)`

If implemented, this method is called when an action is emitted from any Window object associated with the application instance. This method acts as an action listener for all Window objects associated with the application, making it unnecessary to add action listeners to the Window objects manually. `object` defines the object; `label` is its

label defined with `setActionLabel()`; and `action` is the action performed. (See DTComponent `addActionListener()` and `setActionLabel()`.)

`handleWindowClose$(win)`
: If implemented, this method is called when a user clicks the "X" button at the window's top right corner to close it. If `true` is returned, the window is destroyed; if `false`, no change occurs. The return value may be conditionalized as desired, including asking the user for confirmation using `DTQuestion.AskQuestion$()`. If `handleWindowClose$()` is not overridden, the default implementation returns `true`, and calls `exit()` if the window being closed is the last window associated with the application instance.

`removeDialog(tag)`
: Destroys the dialog identified by `tag`.

`removeWindow(arg)`
: Destroys the window identified by `arg`, which may be either a reference to a Window object or a window tag string.

`restoreAfter$()`
: A hook method called after all window drawing in the restore sequence.

`restoreBefore$()`
: A hook method called before all window drawing in the restore sequence.

`restoreDuring$()`
: A hook method called between window chrome drawing and window interior drawing in the restore sequence.

`setStateObject(obj)`
: Sets a reference to the ApplicationStateObject associated with the application instance.

`startup$(rpo)`
: The hook method called during the startup sequence. `startup$()` is the only hook method required to be implemented.

  If the application is being launched with a file, a reference to the RPO that that file points to is supplied in the `rpo` parameter. Otherwise `rpo` is `null`. Applications that do not support file launching may ignore `rpo`.

Functions

`getErrorString(errNum)`
: Converts an error number returned by an Application method into an error message string that may be displayed to users.

DTApplicationManager

The DTApplicationManager package defines the ApplicationManager class.

Classes

None.

Functions

`OpenApp$(appname)`

Opens a new instance of an application. `appname` is a string that specifies the name of the package containing the Application subclass to be instantiated. The name of the Application subclass within this package must match the package name, minus the "App" or "DT" prefix.

Returns an object with properties `success`, `appobj`, and `errno`. If `success` is `true`, `appobj` is a reference to the `AppObject` created by the Application's constructor function. If `success` is `false`, `errno` is an error code that may be compared against the values in `DTApplication.ERRNO` or turned into an error message using `DTApplication.getErrorString()`.

`OpenAppWithArguments$(appname, appargs)`

Opens a new instance of an application, and supplies startup arguments to the application instance. `appname` is a string that specifies the name of the package containing the Application subclass to be instantiated. `appargs` is a reference to an object that contains application-specific information. The application instance may retrieve this information using DTApplication.Application.getAppArguments().

Returns an object with properties `success`, `appobj`, and `errno`. If `success` is `true`, `appobj` is a reference to the `AppObject` created by the Application's constructor function. If `success` is `false`, `errno` is an error code that may be compared against the values in `DTApplication.ERRNO` or turned into an error message using `DTApplication.getErrorString()`.

`OpenFile$(path)`

Opens a file by launching an application and supplying the specified file's RPO in the `rpo` parameter to `DTApplication.Application.startup$()`. `path` specifies the full path to the file to open. The application to be instantiated is chosen automatically based on the type of the RPO contained by the specified file.

Returns an object with properties `success`, `appobj`, `rpo`, and `errno`. If `success` is `true`, `appobj` is a reference to the `AppObject` created by the Application's constructor function, and `rpo` is the RPO object reference of the given path. If `success` is `false`, `errno` is an error code that may be compared against

the values in DTApplication.ERRNO or turned into an error message using DTApplication.getErrorString().

OpenFileWithArguments$(path, appname, appargs)

Opens a file by launching the appropriate application with the optional parameters appname and appargs, either of which may be specified as null. path (required) specifies the full path to the file to open. appname (optional) is a string that specifies the name of the package containing the Application subclass to be instantiated. The name of the Application subclass within this package must match the package name, minus the "App" or "DT" prefix. If appname is omitted, the application to be instantiated is chosen automatically based on the type of the RPO contained by the specified file. appargs (optional) is a reference to an object that contains application-specific information. The application instance may retrieve this information using DTApplication.Application.getAppArguments().

Returns an object with properties success, appobj, rpo, and errno. If success is true, appobj is a reference to the AppObject created by the Application's constructor function, and rpo is the RPO object reference of the given path. If success is false, errno is an error code that may be compared against the values in DTApplication.ERRNO or turned into an error message using DTApplication.getErrorString().

DTApplicationPrefsObject

The DTApplicationPrefsObject package defines the ApplicationPrefsObject class.

Classes

ApplicationPrefsObject

Defines an ApplicationPrefsObject, which is a PersistentObject with an infinite lifetime that is shared among all instances of the application class for a given user. As such, it is a place to store data that affects the behavior of the application, but varies from user to user.

An ApplicationPrefsObject should not be instantiated directly. Instead, the application base class (`DTApplication.Application`) instantiates an ApplicationPrefsObject when the `getPrefs$()` method is called. If an application has its own subclass of ApplicationPrefsObject to manage preferences, an overridden version of `Application.getDefaultPrefsObject()` should instantiate, initialize, and return it.

Inherits from DTObjectStore.RootPersistentObject.

Constructor

`ApplicationPrefsObject(arg1, arg2)`

> `arg1` (optional) defines the user ID, and `arg2` (optional) the ID for this ApplicationPrefsObject.

Methods

None.

Functions

None.

DTApplicationStateObject

The DTApplicationStateObject package defines the ApplicationStateObject class.

Classes

ApplicationStateObject

Defines an ApplicationStateObject, which is a PersistentObject whose lifetime and scope are exactly those of the application instance. It is a place to store any persistent data required to allow the application to retain state across multiple login sessions.

An ApplicationStateObject should not be instantiated directly. Instead, the application base class (`DTApplication.Application`) instantiates an ApplicationStateObject when the `getStateObject()` method is called. If an application has its own subclass of ApplicationStateObject to manage states, it should apply an instance of that subclass using `DTApplication.Application.setStateObject()`.

Inherits from DTObjectStore.PersistentObject.

Constructor

`ApplicationStateObject(arg1, arg2)`

> `arg1` (optional) defines the user ID, and `arg2` (optional) the ID for this ApplicationStateObject.

Methods

None.

Functions

None.

DTAppWindow

The DTAppWindow package defines the AppWindow class: the default window class used by Desktop applications.

Classes

AppWindow

Represents an application window, with title bar, resize handles and an inner pane in which applications display their interface.

An application window is made up of two elements: the frame around the outside, and the application area within this frame. Most methods called on an AppWindow object apply to the inside area. A few methods, however, apply to the entire window. The part to which a given method applies is generally intuitive.

Inherits from DTBorderedWindow.BorderedWindow.

Constructor

`AppWindow()`

Actions

None.

Methods

close()
getApp()
getLocation()
getMenuBar()
getTag()
iconize()
maximize()
setBackgroundColor(color)
setDrawMethod(method)
setLocation(x, y)
setMaxHeight(height)
setMaxWidth(width)
setMenuBar(bool)
setMinHeight(height)
setMinWidth(width)
setResizable(resizable)
setRestoreMethod(method)
setSkipTaskBar(bool)
setTitle(title)
setVisible(isVisible)

`close()`
: Closes the window.

`getApp()`
: Returns a reference to the Application object associated with the window.

`getLocation()`
: Returns the window's pixel coordinate position on the screen. Applies to the entire window, no simply the inside application area.

`getMenuBar()`
: Returns a reference to the menu bar placed in the window.

`getTag()`
: Returns the unique tag string for the window, which is used to differentiate between multiple windows owned by an application.

`iconize()`
: Iconizes the window, hiding it until its taskbar entry is clicked.

`maximize()`
: Maximizes the window size to the size of the desktop.

`setBackgroundColor(color)`
: Sets the background color for the inner portion of the window. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setDrawMethod(method)`
: Sets the method used to define the inner contents of the window. `method` is a string that specifies the name of a method in the application object associated with the window. When this method is called, it will receive one argument, which is a reference to the AppWindow object within which components should be added.

`setLocation(x, y)`
: Sets the window's pixel coordinate position on the screen. Applies to the entire window, not simply the inside application area.

`setMaxHeight(height)`
: Sets the maximum height for the window in pixels. Users will not be able to size the window beyond this height. By default, a window has no maximum height.

`setMaxWidth(width)`
: Sets the maximum width for the window in pixels. Users will not be able to size the window beyond this width. By default, a window has no maximum width.

`setMenuBar(bool)`
: Sets whether the window should provide space for an application menu bar. `true` if space should be provided; `false` if not.

`setMinHeight(height)`
: Sets the minimum height for the window in pixels. Users will not be able to size the window below this height. Default is 10 pixels.

`setMinWidth(width)`
: Sets the minimum width for the window in pixels. Users will not be able to size the window below this width. Default is 10 pixels.

`setResizable(resizable)`

Specifies whether the user is able to resize the application window by dragging one of the sections of the bottom frame. `true` (default) allows the window to be resized; `false` prevents resizing.

`setRestoreMethod(method)`

Sets the method used to define the window's contents during the restore process. If `setRestoreMethod()` is not called, the method set by `setDrawMethod()` is used at both startup and restore.

`setSkipTaskBar(bool)`

Indicates whether an entry should be placed in the Desktop task bar for this window. Default is `false` (to include the entry).

`setTitle(title)`

Sets the `title` string to be displayed for the window in the title bar.

`setVisible(isVisible)`

Sets whether the window is visible. Default is `true`.

Functions

None.

DTArray

The DTArray package defines utility functions for JavaScript arrays. Some of the functions in the DTArray package are redundant with built-in methods of JavaScript arrays, but the versions here are more portable because IE does not support all of the Array methods natively.

Classes

None.

Functions

All functions in DTArray are *global symbols* (see Desktop.com Packages). They may be called without the DTArray qualifier.

`contains(a, e)`

Returns whether array `a` contains element `e`: `true` if it does; `false` if not.

`copy(a)`

Returns a copy of array `a`.

`pop(a)`

Removes and returns the last element of array `a`.

`push(a, e)`

Adds element `e` at the end of array `a`.

`splice(theArray, start, deleteCount, values)`

Inserts and/or deletes elements in an array. A single call to `splice()` may insert items, delete items, or do both simultaneously.

`theArray` is a reference to the array to work with. `start` is an integer that specifies a zero-based index within `theArray` at which `splice()` will begin. `deleteCount` is an integer that specifies how many elements of `theArray` to remove. If `deleteCount` is zero, no deletion is performed. `values` (optional) is an array of elements to insert into `theArray`. If `values` is omitted or an empty array, no insertion is performed. Elements following the location of insertion or deletion are shifted so that no gaps appear in `theArray`, and the `length` property of `theArray` is updated to reflect its new size.

Returns a reference to `theArray`.

For example:

```
var a = ["a", "b", "c"];
     // a == ["a", "b", "c"]
splice(a, 0, 0, ["x"]);
     // a == ["x", "a", "b", "c"]
splice(a, 1, 2, ["p", "q"]);
     // a == ["x", "p", "q", "c"]
splice(a, 2, 1);
     // a == ["x", "p", "c"]
```

Some browsers support a built-in `splice()` method on JavaScript array objects. There are several important differences between the syntax of the DTArray `splice()` function and built-in `splice()` methods. First, the DTArray `splice()` is a function that accepts its target array as a parameter, whereas the built-in `splice()` is a method called directly on array objects. Second, the DTArray `splice()` takes an array of values to insert, while the built-in `splice()` takes an arbitrary number of arguments following `deleteCount`. Third, the two functions return different things.

DTArrayLayoutManager

The DTArrayLayoutManager package defines the ArrayLayoutManager class.

Classes

ArrayLayoutManager

Arranges components in either a single row or single column within a container.

Components are laid out in the order in which they are added to the associated Container. In a horizontal orientation, Components are laid out left to right; in a vertical orientation, they are laid out top to bottom. An ArrayLayoutManager is said to have a *main axis* and an *opposite axis*. For example, in a horizontal orientation, the horizontal (x) axis is the main axis and the vertical (y) axis is the opposite axis.

An ArrayLayoutManager sets the position, and possibly the size, of all child Components in its associated Container. It will never resize its associated Container, as the size of the Container constrains the placement and size of its child Components. The associated Container must always have a defined size, assigned by calling the Container's `setSize()` method.

There are three padding values that may be established for an ArrayLayoutManager: the inter-component padding (`setPadding()` method), the horizontal edge padding (`setHorizontalPadding()` method), and the vertical edge padding (`setVerticalPadding()` method). All three padding values default to zero.

Along the main axis, child Components are positioned such that the space between them exactly equals the inter-component padding specified, and such that the space before the first Component and (if possible) after the last one are both exactly equal to the main-axis edge padding specified. The after-last padding condition is not enforced when all child Components have a fixed main-axis size; see below.

The size of each child Component along the main-axis is determined in one of three ways. If the ArrayLayoutManager's `setSize()` method has been called for the given Component, its main-axis size is set to exactly the size specified in the `setSize()` call. If the ArrayLayoutManager's `setWeight()` method has been called for the given Component, it is a *weighted Component* and its main-axis size is determined as described below. If neither `setSize()` nor `setWeight()` has been called for a Component, it is not resized along the main-axis dimension. Note that *either* `setSize()`, *or* `setWeight()` *or* neither may be called on a component, but not both.

The main-axis size of a weighted Component is set to equal a particular fraction of the *available* space. The available space consists of all pixels along the main axis not used by fixed-size Components and padding. The fraction of this space that a weighted Component receives is equal to the Component's weight divided by the total of all weights.

Along the opposite axis, all child Components are given the same location and resized to the same size. This is done in such a way that the distance between the edge of any Component and

the edge of the Container exactly equals the opposite-axis edge padding specified. For example, a horizontal ArrayLayoutManager attached to a Container of height 30, and with vertical edge padding of 5, will set the height of all child Components to 20 and center them vertically within the Container, leaving 5 pixels of margin along the top and bottom of the Container.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`ArrayLayoutManager(orientation)`

> `orientation` defines the main axis of the container, the axis along which the manager will lay out the components. `orientation` may be `DTArrayLayoutManager.ArrayLayoutManager.VERTICAL` or `DTArrayLayoutManager.ArrayLayoutManager.HORIZONTAL`.

Methods

getHorizontalPadding()	setPadding(padding)
getPadding()	setSize(pixel_size, item)
getVerticalPadding()	setVerticalPadding(p)
setHorizontalPadding(p)	setWeight(weight, item)

`getHorizontalPadding()`

> Returns the horizontal edge padding in pixels. Default is 0.

`getPadding()`

> Returns the padding between components in pixels. Default is 0.

`getVerticalPadding()`

> Returns the vertical edge padding in pixels. Default is 0.

`setHorizontalPadding(p)`

> Sets the horizontal edge padding in pixels. Default is 0.

`setPadding(padding)`

> Sets the padding between components in pixels. Default is 0.

`setSize(pixel_size, item)`

> Sets a fixed main-axis size for a child Component. `pixel_size` is a positive integer specifying the desired size in pixels. `item` (optional) is a reference to the relevant child Component: if omitted, `item` defaults to the most recently added Component.

`setVerticalPadding(p)`

> Sets the vertical edge padding in pixels. Default is 0.

`setWeight(weight, item)`

> Sets a main-axis weight for a child Component. `weight` is a positive number specifying the desired weight. `item` (optional) is a reference to the relevant child Component: if omitted, `item` defaults to the most recently added Component.

Functions

None.

DTAttachmentLayoutManager

Defines the AttachmentLayoutManager class.

Classes

AttachmentLayoutManager

The AttachmentLayoutManager class allows the developer to attach child components to the edge of a Container, or to each other, in flexible ways. By positioning and sizing the child Components within a Container, and possibly resizing the container itself, an AttachmentLayoutManager will automatically satisfy all of the constraints imposed by the attachments.

Attachments are specified for six attachment points of child Components. There are three horizontal attachment points: the LEFT side, the RIGHT side, and the HORIZONTAL center axis; and three vertical attachment points: the TOP side, the BOTTOM side, and the VERTICAL center axis. (Note that attachment points (sides and axes) are specified with all-uppercase strings.)

Every child Component must have an attachment on at least one horizontal and one vertical attachment point. A single attachment in either dimension determines the positioning of the Component in that dimension. A child Component may also have attachments on two attachment points in either or both dimensions. When this is the case, the Component is resized in the relevant dimension so that both attachments are satisfied.

Most attachments are ***fixed attachments***. A side of a Component may be attached to the corresponding edge of the Container, to the corresponding side of a sibling Component, or to the opposite side of a sibling. A center axis of a Component may be aligned with the corresponding center axis of the Container, or the corresponding center axis of a sibling. These fixed attachments all have associated offsets, which specify a margin in pixels between the attachment point and the location to which it is attached. Offsets may be positive, zero, or negative. For side attachments, a positive offset will push a component away from the side to which the attachment applies. For axis attachments, a positive offset will push a component in the positive direction for that axis: to the right for a horizontal axis; down for a vertical.

Attachment layout managers also support ***spring attachments***. Spring attachments must always be paired on opposing sides of a Component: either LEFT and RIGHT; or TOP and BOTTOM. Each spring attachment has an associated *weight* rather than a fixed offset. A Component with spring attachments is positioned such that the ratio of the offsets for the attachments matches the ratio of their spring weights. Spring attachments never cause Component resizing. Spring weights must always be positive.

In some cases, it is possible to specify multiple attachments for a single attachment point, which is called ***ganging***. When ganging is used, the Components will be positioned and sized such that all attachment offsets are *at least* satisfied. In other words, any attachment in a gang may be over-satisfied (offset greater than specified), but none will be under-satisfied. When spring attachments are ganged, all spring weights in the gang must be equal.

If the Container has ***natural sizing*** (see DTComponent), the layout manager (when called) will resize it to exactly fit around its child Components, leaving right and bottom margins

corresponding to the values set with `setHSpacing()` and `setVSpacing()`. Some Components' size or position may depend on the size of the Container, such as those attached to its right or bottom edges or aligned with its center. When the Container is sized to fit around its Components, only those Components that do not depend on the Container's size are considered. If there are no non-dependent Components in a given dimension, the Container is not resized in that dimension.

In calling methods that specify attachments, components and siblings are specified by reference. It is legal to specify a sibling that has not yet been added to the Container, as long as that sibling is added before layout is initially performed.

There must never be circular attachment dependencies; in other words, if component A is attached in a given dimension to its sibling B, B must not be attached to A in the same dimension.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`AttachmentLayoutManager()`

Methods

alignToContainer(component, axis, offset)
alignToSibling(component, axis, sibling, offset)
attachToEdge(component, side, offset)
attachToSiblingOpposite(component, side, sibling, offset)
attachToSiblingSame(component, side, sibling, offset)
setHSpacing(spacing)
setVSpacing(spacing)
springToEdge(component, side, weight)
springToSiblingOpposite(component, side, sibling, weight)
unattach(component, point)

`alignToContainer(component, axis, offset)`

Establishes a fixed attachment with the specified `offset` (optional) along the horizontal or vertical `axis` between the center of the `component` and the center of the Container. If `offset` is omitted, a default of 0 is used.

Attachments set with this method may not be ganged.

`alignToSibling(component, axis, sibling, offset)`

Establishes a fixed attachment with the specified pixel `offset` (optional) along the horizontal or vertical `axis` between the center of the `component` and the center of the specified `sibling`. If `offset` is omitted, a default of 0 is used.

Attachments set with this method may not be ganged.

`attachToEdge(component, side, offset)`

Establishes a fixed attachment with the specified pixel `offset` (optional) between the given `side` of the `component` and the corresponding edge of the Container. If `offset` is omitted, the default value given in `setHSpacing()` or `setVSpacing()` is used.

Attachments set with this method may be ganged with other attachments set with `attachToEdge()` or `attachToSiblingOpposite()`.

`attachToSiblingOpposite(component, side, sibling, offset)`

Establishes a fixed attachment with the specified pixel `offset` (optional) between the given `side` of the `component` and the side of its `sibling` that is opposite `side` (for example, left side to right side). If `offset` is omitted, the default value given in `setHSpacing()` or `setVSpacing()` is used.

Attachments set with this method may be ganged with other attachments set with `attachToSiblingOpposite()` or `attachToEdge()`.

`attachToSiblingSame(component, side, sibling, offset)`

Establishes a fixed attachment with the specified pixel `offset` (optional) between the given `side` of the `component` and the matching side of its `sibling` (for example, left side to left side). If `offset` is omitted, a default of 0 is used.

Attachments set with this method may not be ganged.

`setHSpacing(spacing)`

Sets the default pixel offset for horizontal attachments set with `attachToEdge()` and `attachToSiblingOpposite()`. Default value is 10 pixels.

`setVSpacing(spacing)`

Sets the default pixel offset for vertical attachments set with `attachToEdge()` and `attachToSiblingOpposite()`. Default value is 10 pixels.

`springToEdge(component, side, weight)`

Establishes a spring attachment with the specified `weight` (optional) between the given `side` of the `component` and the corresponding edge of the Container. If `weight` is omitted, a default is used. If this is the first spring attachment for the `side`, the default weight is 1. If not, the default weight is the weight specified in the previous spring attachment for the side.

Attachments set with this method may be ganged with other attachments set with `springToEdge()` or `springToSiblingOpposite()`.

`springToSiblingOpposite(component, side, sibling, weight)`

Establishes a spring attachment with the specified `weight` (optional) between the given `side` of the `component` and the side of its `sibling` that is opposite `side` (for example, left side to right side). If `weight` is omitted, a default is used. If this is the first spring attachment for `side`, the default weight is 1; if not, the default weight is the weight specified in the previous spring attachment for the side.

Attachments set with this method may be ganged with other attachments set with `springToSiblingOpposite()` or `springToEdge()`.

`unattach(component, point)`

Clears all attachments associated with the specified `point` (a side or axis) of `component`. Note that to change an existing attachment, `unattach()` must be called before a new attachment is set. If `unattach()` is not called, the AttachmentLayoutManager will attempt to gang the new attachment with the existing attachment(s).

Functions

None.

DTBasicButton

The DTBasicButton package defines the BasicButton class.

Classes

BasicButton

Defines a button object which may include text, an image, and a background color. The image, if supplied, appears at the left edge of the button. The text, if supplied, is centered in the remaining space.

The BasicButton class differs from the Button class in that the Basic button lacks the highlighted border that Button contains. This makes BasicButton appropriate for use in composite widgets which use Buttons. Most application developers should use Button, instead of BasicButton.

Inherits from DTContainer.Container.

Constructor

`BasicButton(text, icon, toolTip)`

> `text`, `icon`, and `toolTip` define the text, image, and mouse-over text to be associated with the button, respectively. All three are optional. `icon` is an URL string, and takes one of two forms. For an image from the Desktop.com site, `icon` should begin with `top.DTPATH.IMAGES`, and have the specific location appended. For an image that resides in the current user's filesystem, `icon` should be built using `DTFileSystem.makePrivateURL()`. (See DTFileSystem.) `toolTip` sets the ALT text to be displayed during mouse-over events.

Actions

`clicked`: the button was clicked.

Methods

getImage()	setBackgroundColor(color)
getText()	setClicked(isClicked, isOver)
getToolTip()	setImage(icon)
handleMouseEvent(e)	setSize(w,h)
makeTable(isDepressed)	setText(text)
prePaint(buffer)	setToolTip(toolTip)
setAlignment(alignment)	

`getImage()`

> Returns the image used for the button.

`getText()`

> Returns the text used for the button.

`getToolTip()`
: Returns the text that appears when the mouse is over the image (the ALT text). (This method is recognized only by Internet Explorer.)

`setAlignment(alignment)`
: Sets the alignment for the text within the button. Valid input includes `CENTER` (default), `LEFT`, and `RIGHT`.

`setBackgroundColor(color)`
: Sets the background color for the button. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setImage(icon)`
: Sets the image to be used for the button. `icon` is an URL string, as for the DTButton constructor.

`setSize(w, h)`
: Sets the width and height of the button in pixels.

`setText(text)`
: Sets the text string to be used for the button.

`setToolTip(toolTip)`
: Sets the text string to appear when the mouse is over the image (the ALT text). (This method is recognized only by Internet Explorer.)

Note that calling `setSize()`, `setImage()` and `setText()` all require repaints. For best results, call these methods *before* calling `addComponent()`.

Functions

None.

DTBookmarks

The DTBookmarks package defines the DTBookmarks class.

Classes

BookmarkRecord

Defines a Bookmark, with an associated URL and text description.

Inherits from DTObjectStore.RootPersistentObject.

Constructor

`BookmarkRecord(arg1, arg2)`
`arg1` defines the user ID, and `arg2` the Root Persistent Object with which this Persistent Object will be associated.

For more information, see DTObjectStore.RootPersistentObject.

Methods

None.

Functions

`CreateBookmark$(path, url, desc)`
Creates a new BookmarkRecord object and associates it with the given file path. The given URL is parsed to determine whether a matching provider and/or service exists. `path` is the path where the object should be placed. `url` is the URL to associate with the Bookmark object. `desc` is the text string to display to the user as the description for the Bookmark.

Returns the hash: `{success:false, error:<DTFileSystem errno>}` if an error occurred, and `{success:true, bookmarkrecord:<object>}` if the BookmarkRecord was successfully created and saved.

DTBorderedRadioButtonGroup

The DTBorderedRadioButtonGroup package defines the BorderedRadioButtonGroup class.

Classes

BorderedRadioButtonGroup

Defines a group of radio buttons surrounded by a border and a label. The RadioButtonGroup's defining characteristic is that only one button in the group may be selected at a time. In the default state, no item is selected.

A ColumnLayoutManager is automatically attached to the group. It may be manipulated using the `getLayoutManager()` accessor method. (See DTColumnLayoutManager for details.) Radio button groups should not be explicitly sized: calling `setSize()` will cause visual truncation of the group.

BorderedRadioButtonGroups are non-circular Panes. They understand up- and down-arrow keys, tab keys, and the spacebar, as well as mouse events.

See also DTRadioButtonGroup.RadioButtonGroup.

Inherits from DTPane.Pane.

Constructor

`BorderedRadioButtonGroup(text)`

> `text` (optional) is the text string for the group label.

Actions

`changed`: selection has changed to a different item in the group.

Methods

addItem(text, data)
getCheckedItemData()
setAllowUserUncheck(allow)
setCheckedItem(index)
setCheckedItemByData(data)
setLabel(text)
uncheckAllItems()

`addItem(text, data)`

> Adds an option to the group, with the specified `text` string and `data` (optional). Options in the group appear in the order in which they are added. If `data` is absent, `text` is used for the option's data.

`getCheckedItemData()`

> Returns the data associated with the currently selected option, or `null` if no option is selected.

`setAllowUserUncheck(allow)`
: Sets whether the radio button group may be manipulated such that nothing is selected. When `allow` is `true`, users may deselect by clicking a selected option. Default is `false`.

`setCheckedItem(index)`
: Sets the selection within the group by (zero-based) index position.

`setCheckedItemByData(data)`
: Sets the selection in the group, as defined by the data associated with the desired option.

`setLabel(text)`
: Sets the `text` string for the group label.

`uncheckAllItems()`
: Resets the group to the state in which no option is selected.

Functions

None.

DTBorderedWindow

The DTBorderedWindow package defines the BorderedWindow and BorderedWindowFrame classes.

Classes

BorderedWindow

Defines the inner (window) area of a bordered window.

Inherits from DTWindow.Window.

Constructor

None. Should not be instantiated directly, but its methods are inherited by several other object classes.

Actions

None. Actions are delivered from the BorderedWindowFrame associated with the window, and include `closed`, `resized`, `moved`, and `iconized`.

Methods

doneLoading()	setBackgroundColor(color)
getMenuBar()	setMenuBar(bool)
loading()	setTitle(title)

`doneLoading()`

Called when the window is finished loading, just prior to the application calling itself.

`getMenuBar()`

Returns a reference to the menu bar object, or `null` if none exists.

`loading()`

Displays an in-pane method between the painting of the window and the painting of the initial application components.

`setBackgroundColor(color)`

Sets the background color for the inner portion of the window. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setMenuBar(bool)`

Sets whether the window should provide space for an application menu bar. `true` if space should be provided; `false` if not.

`setTitle(title)`
: Sets the `title` string to be displayed for the window in the title bar.

BorderedWindowFrame

Defines the frame for the BorderedWindow.

Inherits from DTWindowFrame.WindowFrame.

Actions

`closed`: the user has clicked on the "x" to close the window.
`resized`: the user has dragged the frame to resize the window.
`iconized`: the user has clicked on the iconize button.
`moved`: the user has dragged the window to move it.

Methods

init() setTitle(title)
setSizeFromDrawable(w, h)

`init()`
: Initializes the window.

`setSizeFromDrawable(w, h)`
: Sets the pixel width and height of the frame, as extending from the inner region of the window.

`setTitle(title)`
: Sets the `title` string to be displayed for the window in the title bar.

Functions

None.

DTBrowser

The DTBrowser package provides functions used to determine what browser is in use.

Classes

None.

Functions

`IsIE(version)`

Returns `true` if the browser is Internet Explorer of at least the numeric `version` specified; `false` if not. (`null` or no argument is equivalent to 0.)

`IsNetscape(version)`

Returns `true` if the browser is Netscape of at least the numeric `version` specified; `false` if not. (`null` or no argument is equivalent to 0.)

DTBrowserWindow

The DTBrowserWindow package defines the BrowserWindow class.

Classes

BrowserWindow

Opens a new browser window and gives it focus. (This class is primarily a wrapper for the native JavaScript `window.open()` method.)

Inherits from DTObjectFramework.DTObject.

Constructor

`BrowserWindow(url)`

`url` (optional) specifies the website to display on launch.

Methods

draw$()
setDelay(msecs)
setFeatures(str)
setHTML(html)
setLocation(x, y)
setMode(str)
setName(name)
setSize(w, h)
setURL(url)

`draw$()`

Creates the new window, using the applied settings. Does not return until the window has been created. Returns a reference to the new window, just as JavaScript's `window.open()` method.

`setDelay(msecs)`

Sets a delay for the window's appearance in milliseconds. Due to the event model, new browser windows have a tendency to appear behind the Desktop window. Setting a delay solves this problem, and displays the window in its proper place on top. (The default delay of 100ms works well with both IE4 and IE5.)

`setFeatures(str)`

Sets the features used to call `window.open()`. Any features without a corresponding method function should be included here. By default, this string reads: `location=yes, menu=yes, toolbar=yes, scrollbars=yes, resizable=yes, status=yes`.

`setHTML(html)`

Sets the `html` to be the window's content. (An alternative to `setURL()`.)

`setLocation(x, y)`

Sets a screen location for the top left corner of the window, overriding `setMode()`. (This method works in IE5 and NN, but not in IE4.)

`setMode(str)`

Sets the display mode for the window, which may be `auto` (the default) or `normal`. In `auto` mode, the window will appear slightly smaller than the Desktop window, and centered within it. In `normal` mode, the window's size and position will be determined by browser defaults.

`setName(name)`

Sets the text string to display in the window's title bar.

`setSize(w, h)`

Sets a width and height for the window, overriding `setMode()`.

`setURL(url)`

Sets an initial URL for the window to display.

Functions

None.

DTButton

The DTButton package defines the Button class.

Classes

Button

Defines a button object which may include text and an image. The image, if supplied, appears at the left edge of the button. The text, if supplied, is centered in the remaining space.

Inherits from DTContainer.Container.

Constructor

`Button(text, icon, toolTip)`

> `text`, `icon`, and `toolTip` define the text, image, and mouse-over text to be associated with the button, respectively. All three are optional. `icon` is an URL string, and takes one of two forms. For an image from the Desktop.com site, `icon` should begin with `top.DTPATH.IMAGES`, and have the specific location appended. For an image that resides in the current user's filesystem, `icon` should be built using `DTFileSystem.makePrivateURL()`. (See DTFileSystem.) `toolTip` sets the ALT text to be displayed during mouse-over events.

Actions

`clicked`: the button was clicked.

Methods

getImage()	setBackgroundColor(color)
getText()	setImage(icon)
getToolTip()	setText(text)
setAlignment(alignment)	setToolTip(toolTip)

`getImage()`

> Returns the image used for the button.

`getText()`

> Returns the text used for the button.

`getToolTip()`

> Returns the text that appears when the mouse is over the image (the ALT text). (This method is recognized by Internet Explorer only.)

`setAlignment(alignment)`

> Sets the alignment for the text within the button. Valid input includes `CENTER` (default), `LEFT`, and `RIGHT`.

`setBackgroundColor(color)`

Sets the background color for the button. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setImage(icon)`

Sets the image to be used for the button. `icon` is an URL string, as for the DTButton constructor.

`setText(text)`

Sets the text string to be used for the button.

`setToolTip(toolTip)`

Sets the text string to appear when the mouse is over the image (the ALT text). (This method is recognized only by Internet Explorer.)

Note that calling `setSize()`, `setImage()` and `setText()` all require repaints. For best results, call these methods *before* calling `addComponent()`.

Functions

None.

DTCallback

The DTCallback package defines the Callback class.

Classes

Callback

Defines a wrapper for a function or method to be invoked at a later time, potentially more than once. A Callback object manages three things: a *target* to be invoked, a *creator data* parameter, and a *caller data* parameter.

A Callback object's target is established when the Callback object is instantiated. The target is either a function reference or an object-reference/method-name pair. Blocking functions and methods cannot be used as Callback targets.

Creator data and caller data are parameters that are passed to the target when it is invoked. Both are optional, and may store values of any type. Creator data is established when the Callback is instantiated, and does not vary from invocation to invocation. Caller data is established when the Callback is invoked, and may vary from invocation to invocation.

The Callback class is a global symbol, meaning that Callback objects may be instantiated without reference to the DTCallback package.

See also DTNamedCallback.

Inherits from DTObjectFramework.DTObject.

Constructor

`Callback(arg1, arg2, arg3)`

Constructs a Callback object.

To create a Callback to a function, pass a reference to the target function for `arg1`. Creator data may be passed for `arg2` (optional).

To create a Callback to a method, pass a reference to the object containing the target method for `arg1`, pass a string containing the name of the target method for `arg2`. Creator data may be passed for `arg3` (optional).

Methods

`call(arg)`

Invokes the Callback. `arg` (optional) provides caller data.

Functions

None.

DTCheckBox

The DTCheckBox package defines the CheckBox class.

Classes

CheckBox

Defines a single checkbox, which may include a text label extending to the right.

Inherits from DTContainer.Container.

Constructor

`CheckBox(text, checked)`

> `text` (optional) specifies the text string for the CheckBox. `checked` (optional) specifies whether the CheckBox is initially checked: `true` if checked, `false` if not. If absent, the CheckBox will be initially unchecked.

Actions

`checked`: the CheckBox has changed to the checked state.
`unchecked`: the CheckBox has changed to the unchecked state.

Methods

getChecked()	setChecked(checked)
getText()	setText(text)

`getChecked()`

> Returns the state of the CheckBox: `true` if checked; `false` if not.

`getText()`

> Returns the text string associated with the CheckBox.

`setChecked(checked)`

> Sets the state of the CheckBox: `true` if checked; `false` if not.

`setText(text)`

> Sets the text string associated with the CheckBox.

Functions

None.

DTColumnLayoutManager

The DTColumnLayoutManager package defines the ColumnLayoutManager class.

Classes

ColumnLayoutManager

A layout manager that arranges child Components in a specified number of columns. Components flow down, then across, in the order in which they were added. Items are arranged roughly to minimize the height of the columns.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`ColumnLayoutManager()`

Methods

setColSpacing(spacing) setVertSpacing(spacing)
setNumCols(n)

`setColSpacing(spacing)`
: Sets the number of pixels separating columns horizontally. Default is 20.

`setNumCols(n)`
: Sets the number of columns. Default is 1.

`setVertSpacing(spacing)`
: Sets the number of pixels separating components vertically. Default is 5.

Functions

None.

DTComponent

The DTComponent package defines the Component class.

Classes

Component

Defines a generic component. Serves as the base class for all GUI components. May be instantiated directly to create a component whose appearance is to be defined by HTML content.

Inherits from DTContent.Content.

Constructor

`Component()`

Actions

`resized`: the Component has been resized.

`moved`: the Component has been moved.

Methods

addActionListener(o, m)	setActionLabel(label)
addMouseListener(o, m)	setCursor(pointer_img)
getActionLabel()	setEnabled(enabled)
getScreenLocation()	setFocusable(focusable)
getSizing()	setHeight(h)
isActive()	setLocation(x, y)
isEnabled()	setPrimary()
removeActionListener(o, m)	setSize(w, h)
removeMouseListener(o, m)	setSizing(s)
repaint(now)	setWidth(w)

`addActionListener(o, m)`

Adds an action listener to the Component. An action listener is a method that is called each time the Component emits an action. The actions emitted by each Component class are described in its "Actions" section. A Component may have any number of action listeners, and the order in which they are called is undefined.

`o` is a reference to the object in which the listener method is to be called, and `m` is the name of the method. The listener method should be defined to accept three parameters: `obj`, a reference to the Component emitting the action; `label`, the Component's action-label string; and `action`, a string that specifies what action is occurring.

`addMouseListener(o, m)`

Adds a mouse listener to the Component. A mouse listener is a method that is called each time a mouse event occurs within the Component. A Component may have any number of mouse listeners, and the order in which they are called is undefined.

`o` is a reference to the object in which the listener method is to be called, and `m` is the name of the method. The listener method should be defined to accept one parameter: `event`, a reference to a DTEventObject.EventObject that describes the mouse event.

Note that mouse listeners provide a lower-level view of user activity than action listeners. Most Component classes are designed to emit actions each time an event occurs, including each time a mouse event occurs. Therefore, it should rarely be necessary to use a mouse listener in addition to normal Component action listeners.

`getActionLabel()`

Returns the action label string set with `setActionLabel()`.

`getScreenLocation()`

Returns an object with properties `x` and `y`, whose integer values describe the location of the Component relative to the Desktop.com screen (the browser window in which the Desktop.com environment is running). This method differs from `getLocation()`, which returns the coordinates of a Component relative to its parent Container.

`getSizing()`

Returns the current sizing style.

`isActive()`

Returns `true` if the Component is active, `false` if inactive. (Being active is a focus-related concept.)

`isEnabled()`

Returns `true` if the Component is enabled, `false` if not.

`removeActionListener(o, m)`

Removes an action listener added with `addActionListener()`.

`removeMouseListener(o, m)`

Removes a mouse listener added with `addMouseListener()`.

`repaint(now)`

Redraws the Component by regenerating its on-screen HTML content. If `now` is `true`, repainting occurs immediately. If `now` is `false` or omitted, repainting is scheduled to occur sometime in the near future. It is often best to delay `repaint()`, as in some cases multiple requests may be made to repaint the same Component; if these requests are for delayed repainting, only a single redraw will occur.

`setActionLabel(label)`

Sets a string label for the component that will appear as a parameter in calls to action listeners, providing a way to distinguish between multiple components that call the same action listener. (See `addActionListener()` above.) Each component subclass has its own, default action label; this default usually matches the name of the class.

`setCursor(pointer_img)`

Sets the image for the cursor to show when it is over the Component. `pointer_img` is a string that takes one of the following values: `"auto"`, `"crosshair"`, `"default"`, `"e-resize"`, `"help"`, `"move"`, `"n-resize"`, `"ne-resize"`, `"nw-resize"`, `"pointer"`, `"s-resize"`, `"se-resize"`, `"sw-resize"`, `"text"`, or `"wait"`. Default value is `"default"`.

(Supported only in Internet Explorer. Equivalent to the `style.cursor` property in the Internet Explorer DOM: consult a DHTML reference for details.)

`setEnabled(enabled)`

Enables the Component if `enabled` is `true`; disables it if `enabled` is `false`. A disabled Component does not respond to mouse or keyboard events, and may have a different appearance than when it is enabled. Components are enabled by default.

`setFocusable(focusable)`

Sets whether the component may receive focus. Default is `true`.

`setHeight(h)`

Changes the height of the component to the defined pixel size. Generates the action `resized`.

`setLocation(x, y)`

Moves the component to the defined location in its container. Generates the action `moved`.

`setPrimary()`

Makes the specified Component the primary Component on the screen. (Being primary is a focus-related concept.)

`setSize(w, h)`

Changes the width and height of the component to the defined pixel dimensions. Generates the action `resized`.

`setSizing(s)`

Sets the sizing style for the Component. The sizing style is a hint to layout managers regarding how a Component may be resized during the layout process. Different layout managers interpret the sizing style in different ways.

Valid sizing styles are `DTComponent.Component.SIZING_NATURAL`, which indicates that a Component should be allowed to assume the size dictated by its content;

`DTComponent.Component.SIZING_WIDTH`, which indicates that a Component's width may be changed but its height should remain unchanged; and `DTComponent.Component.SIZING_BOTH`, which indicates that a Component may be resized in both dimensions.

The default sizing style is SIZING_NATURAL.

`setWidth(w)`
: Changes the width of the component to the defined pixel size. Generates the action `resized`.

Functions

None.

DTConfirm

The DTConfirm package defines the `DoConfirm$()` function.

Classes

None.

Functions

`DoConfirm$(text, where)`

Displays a confirmation dialog window, similar to JavaScript's `confirm()`, with the specified `text` string, and OK and Cancel buttons. `where` (optional) specifies a reference to an object (either an Application or a Window) relative to which the dialog is to be centered and made modal. If `where` is omitted, the dialog will be centered on the screen and made system-modal.

`DoConfirm$()` should always be called as a blocking function. It will not return until the user clicks the OK or Cancel button in the dialog displayed. The return value will indicate which button was pressed: `true` for the OK button, and `false` for the Cancel button.

`DoConfirm$()` is preferable to the JavaScript `confirm()` because it is implemented within the Desktop.com environment, which allows more control over appearance and provides the user with feedback that clearly comes from within Desktop.com.

Dialog windows are resized to fit the text. `text` is wrapped where appropriate, may include HTML tags, and will translate `"\n"` appearing in text as a line break.

DTContainer

The DTContainer package defines the Container class.

Classes

Container

Defines a generic container. By default, a container does not include a layout manager.

Inherits from DTComponent.Component.

Constructor

`Container(layout_manager)`

> `layout_manager` (optional) is a reference to the LayoutManager object to be attached to the container.

Actions

None.

Methods

addComponent(component, constraints)
doLayout(force)
getComponent(index)
getLayoutManager()
getNumberComponents()
removeComponent(c)
removeComponents()
setLayoutManager(lm)
setSize(w, h)

`addComponent(component, constraints)`

> Adds the specified component to the container. `constraints` is the constraint object to be passed to the layout manager's `addComponent` method. Note that not all layout managers require a constraints object.

`doLayout(force)`

> Asks the layout manager to recalculate layout.

`getComponent(index)`

> Returns a reference to the child Component with the specified `index` from the components array of the container. (Index numbers are generated for components by the order in which they are added to a container.)

`getLayoutManager()`

> Returns the LayoutManager for this Container, or `null` if none exists.

`getNumberComponents()`

> Returns the number of child components in this container.

`removeComponent(c)`
: Removes a component from the container. c is a reference to the component to remove.

`removeComponents()`
: Removes all components from the container.

`setLayoutManager(lm)`
: Sets the layout manager. `lm` is a reference to the layout manager to set. Any components added before this method will be ignored by the layout manager.

`setSize(w, h)`
: Specifies the width and height dimensions of the container. Causes the LayoutManager (if any) to recalculate layout.

Functions

None.

DTContent

The DTContent package defines the Content class.

Classes

Content

The Content class is the browser-specific class that represents a region of dynamic HTML content in the browser.

Never construct a raw Content instance. Instead, use `DTComponent.Component`, a browser-neutral subclass of DTContent.

Inherits from DTObjectFramework.DTObject.

Constructor

`DTContent()`

Methods

getBackgroundColor()	setBackgroundColor(color)
getBackgroundImage()	setBackgroundImage(image_location)
getContent()	setContent(content, style)
getImages()	setHeight(h)
getLocation()	setLocation(x, y)
getSize()	setSize(w, h)
getStyle()	setStyle(style)
getVisible()	setVisible(desiredVisibility)
getZ()	setWidth(w)
IsOnScreen()	setZ(z)

`getBackgroundColor()`
: Returns the background color for the region.

`getBackgroundImage()`
: Returns the background image for the region.

`getContent()`
: Returns the `content` (as an HTML string) and an optional `style` (a reference to a `DTStyle.Style` object) of the region.

`getImages()`
: Returns the array of images for the region. The order of entries in the array corresponds to the order in which <IMG> elements appear in the region's content. Each entry in the array is an IMG object from the browser DOM. (Consult a DHTML reference for details.)

The array contains only those <IMG> elements from the content of the region itself; it excludes <IMG> elements from child regions.

The main purpose of calling `getImages()` is to change an image dynamically. For example:

```
this.getImages()[0].src=top.DTPATH.IMAGES+"foo.gif"
```

`getLocation()`
: Returns the location of the region relative to its parent Container.

`getSize()`
: Returns an object with two properties: `w`, the width, and `h`, the height of the region in pixels.

`getStyle()`
: Returns the style of the region as an instance of `DTStyle.Style`. If there is no current style object for the region, one is created and returned.

`getVisible()`
: Returns whether the region is visible. Default is `true`.

`getZ(z)`
: Returns the zero-based stacking position of the region within its parent Container.

`IsOnScreen()`
: Returns `true` if the region is painted in the browser; `false` if not.

`setBackgroundColor(color)`
: Sets the background color for the region. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setBackgroundImage(image_location)`
: Sets the background image for the region. `image_location` is an URL string, and takes one of two forms. For an image from the Desktop.com site, `icon` should begin with `top.DTPATH.IMAGES`, and have the specific location appended. For an image that resides in the current user's filesystem, `icon` should be built using `DTFileSystem.makePrivateURL()`. (See DTFileSystem.)

`setContent(content, style)`
: Sets the `content` (as an HTML string) and an optional `style` (a reference to a `DTStyle.Style` object) of the region.

`setHeight(h)`
: Sets the height of the region in pixels.

`setLocation(x, y)`
: Sets the location of the region relative to its parent Container.

`setSize(w, h)`
: Sets an object with two properties: `w`, the width, and `h`, the height of the region in pixels.

`setStyle(style)`
: Sets the style of the region as an instance of `DTStyle.Style`. If the style object is not supplied, a default (empty) style is used.

`setVisible(desiredVisibility)`
: Sets whether the region is visible: `true` if visible, `false` if not. Default is `true`.

`setWidth(w)`
: Sets the width of the region in pixels.

`setZ(z)`
: Sets the zero-based stacking position of the region within its parent Container.

Functions

None.

DTDetailsRow

The DTDetailsRow package defines the DetailsRow class.

Classes

DetailsRow

Defines a table row of text, which includes an icon to the far left. DetailsRow differs from a table in that the spacing, alignment and width of the cells of text may be set individually.

Note that for efficiency, `repaint()` is not called after changing text, icons or any of the content. To see changes, call `repaint()` on this object after making the changes.

The number of columns exclude the icon at the left of the row.

Inherits from DTComponent.Component.

Constructor

`DetailsRow(column_text, width, icon, alignments)`

`column_text` is an array of strings which specify the text of each column in the table; `width` is an array of pixel values which specify their widths. `icon` is a string URL which sets the icon to be displayed before the first column. `alignments` is an array of symbolic constants which specify the column/text alignments: possible values include `DTDetailsRow.DetailsRow.ALIGN_CENTER`, `DTDetailsRow.DetailsRow.ALIGN_LEFT`, and `DTDetailsRow.DetailsRow.ALIGN_RIGHT`.

Actions

None.

Methods

getText(col_num) setIcon(icon)
setColumnPadding(padding) setText(col_num, text, alignment, width)
setColumnWidth(width)

`getText(col_num)`

Returns the text from the column specified by its zero-based column number.

`setColumnPadding(padding)`

Sets the number of pixels between columns.

`setColumnWidth(width)`

Sets the width of the columns in pixels. `width` is an array of integers.

`setIcon(icon)`

Sets the icon to be displayed before the first column of the row. `icon` is an URL, as defined for the DTDetailsRow constructor.

`setText(col_num, text, alignment, width)`

Sets the text string to be displayed in the column specified by its zero-based column number. `alignment` is a symbolic constant, as defined for the DTDetailsRow constructor `alignments` parameter, which sets the alignment for the text within the column. `width` defines the width of the text within the column, in pixels.

Functions

None.

DTDialogWindow

The DTDialogWindow package defines the DialogWindow class.

Classes

DialogWindow

Defines a dialog window. Dialog windows are similar to Application windows, except that while in existence, they maintain focus, and are not resizable.

Inherits from DTBorderedWindow.BorderedWindow.

Constructor

```
DialogWindow()
```

Actions

None.

Methods

None. (All methods are inherited.)

Functions

None.

DTDragManager

The DTDragManager package enables drag and drop both within and between Desktop applications. Enabling drag for a component causes rubber-banding of the component: an outline of the component is created when the user holds the mouse down when above it, which follows the movement of the cursor. When the mouse is released, DragManager checks to see if it is over an interested target, and notifies that target if it is.

The DragManager does not check the visibility of a container before firing the drop event, nor does it check to see if that container is blocked by an arbitrary component (or even clipped by a parent container) on the screen. Doing this would require the DragManager to examine the location of every component on the screen, which would be prohibitively expensive.

To verify the validity of a drop event, the DragManager checks to see if the event is blocked by a window, then leaves any additional checks to the application developer (the owner of the window). Additional checks are necessary only if the droppable component could be blocked or clipped by some other component (or made invisible).

The DragManager does not perform any actions implied or necessitated by a drag and drop procedure. It simply notifies the drop Component of the event. The involved Components must then handle any action subsequent to the drop themselves, such as moving the component, and sending re-parenting notification to the appropriate Containers.

Classes

None.

Functions

`DisableDrag(component)`

Disables dragging for the specified component.

`DisableDrop(component, object, method)`

Unregisters the specified component as an interested drop target.

`EnableDrag(component)`

Enables dragging for the specified component. The Drag Manager then listens for mouse events and creates and moves the component outline.

To make componentA draggable, use:
`DTDragManager.EnableDrag(componentA)`

`EnableDrop(component, object, method)`

Registers a component as an interested drop target. When a drag event ends on such a component, the DragManager sends notification of the event by passing the component being dragged as an argument to the specified `object` and `method`.

To make componentB available to drag-n-drop events, use:
`DTDragManager.EnableDrop(componentB, object, method)`

Then, if any draggable component (such as componentA) is dragged and dropped on componentB, `object[method]()` is called, with the first argument as `componentB` and the second argument as `componentA`. `method` is `dropPerformed` by default.

DTDropDownComboBox

The DTDropDownComboBox package defines the DropDownComboBox class.

Classes

DropDownComboBox

Defines a text input field with a drop-down combo box, in which users may either type in the value or select among a list of drop-down menu items.

Inherits from DTPane.Pane.

Constructor

`DropDownComboBox(index)`

> `index` (optional) specifies the default selection. If absent, the default selection will be zero (the first item in the Menu).

Actions

`changed`: The user has changed the selection. Call `getIndex()` or `getText()` to determine which is the current selection.

Methods

addItem(text, index, id)	setEditable(bool)
getIndex()	setIndex(index)
getSelectedId()	setSelectedId(id)
getText()	setText(text)
modifyText(id, text)	setWidth(w)
removeId(id)	size()

`addItem(text, index, id)`

> Adds an item to the list at the specified (zero-based) index location, with the specified `text` and `id`.

`getIndex()`

> Returns the (zero-based) index of the current selection.

`getSelectedId()`

> Returns the ID of the current selection.

`getText()`

> Returns the text string of the current selection, or the currently visible text.

`modifyText(id, text)`

> Changes the display-text of the item with the specified `id` to `text`.

`removeId(id)`
: Removes the item with the specified `id`.

`setEditable(bool)`
: Sets whether the text input field is editable: `true` if editable (default); `false` if not.

`setIndex(index)`
: Sets the current selection by (zero-based) `index`. The `changed` action is not emitted.

`setSelectedId(id)`
: Sets the selected item's ID to `id`. Pass in `null` to reset the menu to the first option in the list.

`setText(text)`
: Sets the text string displayed in the text field to `text`.

`setWidth(w)`
: Sets the width of the DropDownComboBox in pixels.

Functions

None.

DTEventGrabber

The DTEventGrabber package defines the EventGrabber class.

Classes

EventGrabber

Defines an event grabber object, which is an invisible component that may be sized and placed over other components in order to intercept events and translate them into actions. An event grabber may be placed over components that do not normally generate actions, in order to create the illusion that they do. This allows developers to add events and actions as desired.

An event grabber may also be placed over multiple Components, to be used for unified mouse listening. It will emit a single event, with pixel coordinates, for all mouse events which take place within it. If the coordinates for all Components are known, the event may then be processed for the appropriate Component, without attaching an event listener to each Component individually. This feature is also useful when only a single action is required from multiple components.

The component generates `clicked` and `doubleclicked` actions.

Inherits from DTComponent.Component.

Constructor

`EventGrabber()`

Actions

`clicked`: the user has clicked on the EventGrabber.
`doubleclicked`: the user has double-clicked on the EventGrabber.

Methods

None. (Methods allowing manipulation are inherited from content.)

Functions

None.

DTEventObject

The DTEventObject package defines the EventObject class.

Classes

EventObject

Defines a structure that provides details of a mouse or keyboard event. Unlike most classes in the DTAPI, EventObject exposes properties rather than methods.

Constructor

None. (This class should not be instantiated directly.)

Properties

alt	offY
component	original
ctrl	shift
key	type
mButton	x
offX	y

`alt`

A Boolean indicating whether the Alt key was down at the time of the event: `true` if pressed, `false` if not. Present only for keyboard events.

`component`

A reference to the Component in which the event occurred. If the event is being redistributed to a parent Component, `component` is a reference to the immediate child Component passing the event up.

`ctrl`

A Boolean indicating whether the Ctrl key was down at the time of the event: `true` if pressed, `false` if not. Present only for keyboard events.

`key`

The key that was pressed. Present only for keyboard events. Symbolic constant that may be compared against the constants defined in DTKeyEvent. (See also Appendix II: KeyEvent constants.)

`mButton`

An integer specifying which mouse button was pressed. Present only for `mousedown`, `mouseup`, and `click` events. For a two-button mouse, left button is `1`, and right is `2`. For a three-button mouse, left is `1`, right is `3`, and center is `2`.

`offX`

The x coordinate of the mouse pointer at the time of the event, relative to `component`. Present only for mouse events.

`offY`

The y coordinate of the mouse pointer at the time of the event, relative to `component`. Present only for mouse events.

`original`

A reference to the Component in which the event originally occurred.

`shift`

A Boolean indicating whether the Shift key was down at the time of the event: `true` if pressed, `false` if not. Present only for keyboard events.

`type`

A string indicating the type of event that has occurred. Possible values are `"mousemove"`, `"mousedown"`, `"mouseup"`, `"click"`, `"keydown"`, `"keyup"`, and `"keypress"`.

`x`

The x coordinate of the mouse pointer at the time of the event, relative to the entire browser window. Present only for mouse events.

`y`

The y coordinate of the mouse pointer at the time of the event, relative to the entire browser window. Present only for mouse events.

Functions

None.

DTFileDownload

The DTFileDownload package allows users to select and download files from their account to their local computer.

Classes

None.

Functions

`download$(path_or_app)`

> Downloads a file from the user's filesystem to their local computer. If the `path_or_app` argument is a string, it is assumed to be a path to the file to be downloaded, and download begins immediately. If `path_or_app` is a reference to an application object, the `DTFileWindow.GetFile$()` is called to allow the user to select the file to downloaded.

DTFileSharing

The DTFileSharing package provides a layer on top of the filesystem that allows users to share directories with other users, and to access others' shared files.

File sharing is based on two mechanisms: shared navigation and sharing permissions.

Shared navigation enables a user to specify a path into another user's filesystem space. Paths within a user's space are relative to the filesystem root. Paths that reference another user's space are relative to that user's *share root*. A user's share root is a publicly visible directory that contains *sharing links*, which are smart links to the directories that the user has defined as shared. A path into another user's space has the syntax `"user:/sharename/path"`, where `user` is the target user's username, `sharename` is the name of the share link to follow in the target user's share root, and `path` is a path within the directory to which `sharename` points.

Sharing permissions are attributes of a sharing link. Sharing permissions are embodied in access control lists, or ACLs. An ACL defines which users the server will allow to follow a sharing link.

A directory is said to be *directly shared* if it is the target of a sharing link. A directory or file is said to be *indirectly shared* if it has an ancestor directory that is the target of a sharing link.

See also DTFileSystem.

All paths are strings of link names separated by slash characters. For more information on paths and links, see the Developers' Guide chapter: The FileSystem.

Classes

AccessControlList

Encapsulates access permissions on a node in the filesystem. An AccessControlList has one property per user to whom access is granted.

Inherits from DTObjectStore.PersistentObject.

Constructor

None. (Only `createACL$()` should instantiate this class.)

Methods

`setPermission(key, value)`

A wrapper around the `PersistentObject.set()` method which checks that a valid permission value is supplied. `key` is the username as a string, and `value` the permission allowed, which may be "R" (read), or "RW" (read-write). The special username "`all`" denotes permissions for all users.

Functions

Error Returns

In the DTFileSharing package, all blocking functions (i.e., all functions whose names end in "$") return objects. These objects have a variety of properties, which include properties that match the blocking function's argument(s), and a Boolean `success` property that indicates whether the call succeeded. When `success` is `false`, there is an `error` property whose value is an error code. Error codes may be compared against properties of `DTFileSharing.ERRNO` (e.g., `DTFileSharing.ERRNO.ERR_FILENOTFOUND`), or they may be converted to meaningful error messages using `DTFileSharing.getErrorString()`. When `success` is `true`, certain calls place additional properties in the objects they return. These properties are noted as bracketed return property names in the documentation of each blocking function.

For example, the object returned from `getACL$(path): {path, success, [error], [acl]}` has a `path` property whose value matches the argument passed to `getACL$()`. The `success` property is always present. When `success` is `false`, `error` is present; when `success` is `true`, `acl` is present. In this case, `acl` is a reference to the object retrieved by `getACL$()`.

The list of definitions for the errors returned is given at the end of the functions list.

`createACL$(path)`

Creates and returns a new AccessControlList (ACL) object for a given sharing link under the current user's share root. `path` is a string which specifies the sharing link on which the ACL is to be placed, and will be interpreted as relative to the share root. If an ACL already exists for that sharing link, it will simply be returned.

Returns `{path, success, [acl, existing], [error]}`, where `acl` is a reference to the new ACL object, and `existing` is a Boolean that indicates whether the ACL already existed.

Possible errors include: `ERR_INDIRECT`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOPERMS`.

`getACL$(path)`

Obtains the AccessControlList object, if any, associated with a given sharing link under the current user's share root. `path` will be interpreted as relative to the share root.

Returns `{path, success, [error], [acl]}`. `acl` is `null`.

Possible errors include: `ERR_BADPATH`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOPERMS`.

`getErrorString(err)`

Converts an error number returned by a DTFileSharing function into an error message string that may be displayed to users.

`getShareStatus$(path)`

Returns the sharing status of the node specified, by `path`, in the private portion (anywhere not under the share root) of the user's filesystem. `path` is relative to the filesystem root.

Note that `getShareStatus$()` is a potentially expensive operation, as it may need to traverse many filesystem nodes.

Returns `{path, success, [sharing], [error]}`, where `sharing` is `null` if the node is not shared. If the node is shared, sharing is an object with one or both of `{direct, indirect}`, each an array of objects `{path, acl}`, with `acl` an AccessControlList.

Possible errors include: `ERR_BADPATH`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOTPRIVATE`.

`getShareTarget$(sharePath)`

Returns the target of the sharing link specified by `sharePath` under the current user's share root. `sharePath` is relative to the share root.

Returns `{sharePath, success, [error], [targetPath]}`

Possible errors include: `ERR_BADPATH`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOTSHARELINK`.

`removeACL$(path)`

Destroys an AccessControlList on a sharing link under the current user's share root. The `path` argument is interpreted in the same way as the argument to `createACL$()`.

Returns `{path, success, [error]}`

Possible errors include: `ERR_BADPATH`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOPERMS`.

`share$(sharePath, target)`

Creates a sharing link from the current user's share root to a directory in the current user's filesystem. Calling `share$()` is the first of two steps necessary to share a directory; setting up an ACL with `createACL$()` is the second.

The `sharePath` argument specifies the path to the link to be created under the share root. It will be interpreted as relative to the share root. `target` specifies a path to the directory being shared, relative to the filesystem root.

Returns {sharePath, target, success, [error]}

Possible errors include: ERR_BADARGS, ERR_BADPATH, ERR_INDIRECT, ERR_NOPATH, ERR_NOTDIR, ERR_EXISTS, and ERR_NOSMART.

unshare$(target)

Destroys a sharing link under the current user's share root. The link to be destroyed may be specified either by source or by target. To specify a source, provide a path that resolves to a sharing link under the share root. In this case, only the specified sharing link is unlinked. To specify a target, provide a path that resolves to a directory that is not under the share root. In this case, all sharing links that point to the specified directory are unlinked. To specify a source, use the syntax ":/path/to/link," which will be interpreted as relative to the share root. To specify a target, omit the colon from your path, and the path will be interpreted as relative to the filesystem root.

Possible errors include: ERR_BADARGS, ERR_BADPATH, ERR_INDIRECT, ERR_NOPATH, ERR_NOTDIR, and ERR_NOSMART.

Errors

ERR_BADARGS: invalid argument(s).

ERR_BADPATH: syntax error in path.

ERR_EXISTS: there is already something at the specified location.

ERR_INDIRECT: sharePath traverses a share link, and thus does not point to a shared folder.

ERR_NOPATH: path does not exist.

ERR_NOPERMS: path points at something that cannot have an ACL.

ERR_NOSMART: the target is of a type that cannot be shared.

ERR_NOTAVAIL: specified user has no shared area, or specified path points to a symlink that contains a private path.

ERR_NOTDIR: path attempts to traverse a non-directory.

ERR_NOTEMPTY: specified folder is not empty. unlink() or unshare() everything in it first.

ERR_NOTFILE: path points to something that is not a file.

ERR_NOTPRIVATE: path points at something in the shared area. This method is meant only for things outside the shared area.

ERR_NOTSHARELINK: path points at something that is not a share link.

ERR_NOUSER: no user with specified username.

DTFileSystem

The DTFileSystem package defines a collection of functions that provide a persistent filesystem hierarchy, within which persistent data may be stored.

The filesystem consists of nodes, which may be files or directories. Files serve as placeholders for RootPersistentObjects (RPOs). Directories serve to organize files.

A *hard link* points from a directory node to a node (file or directory) that it contains, and is the primary link to a node. A *smart link* also points from a directory to a node that it contains, but is viewed as a secondary link to a node, and may define secondary routes between directories and/or files in a user's filesystem. A symbolic link, or *symlink,* is a path reference to another node in which only the target node's path is encoded. By convention, symlinks are used only to create links to other users' filesystems.

For more information, see The DTFileSystem.

Classes

None.

Functions

Error Returns

In the DTFileSystem package, all blocking functions (i.e., all functions whose names end in "$") return objects. These objects have a variety of properties, which include properties that match the blocking function's argument(s), and a Boolean `success` property that indicates whether the call succeeded. When `success` is `false`, there is an `error` property whose value is an error code. Error codes may be compared against properties of `DTFileSystem.ERRNO` (e.g., `DTFileSystem.ERRNO.ERR_FILENOTFOUND`), or they may be converted to meaningful error messages using `DTFileSystem.getErrorString()`. When `success` is `true`, certain calls place additional properties in the objects they return. These properties are noted as bracketed return property names in the documentation of each blocking function.

For example, the object returned from `get$(path): {path, success, [error], [object]}` has a `path` property whose value matches the argument passed to `get$()`. The `success` property is always present. When `success` is `false`, `error` is present; when `success` is `true`, `object` is present. In this case, `object` is a reference to the object retrieved by `get$()`.

The list of definitions for the errors returned is given at the end of the functions list.

In general, a path passed to any DTFileSystem function may traverse any kind of link: hard link, smart link, or symlink. Except where noted, all paths must be absolute.

The paths passed to `get$()`, `getDirectoryEntryArray$()`, `getDirectoryEntries$()`, and `getEntry$()` may be paths within the current user's filesystem, or they may refer to files in other users' filesystems. To refer to another user's filesystem, construct a path string of the form "username:/path/to/file. Such paths are interpreted as relative to the named user's share root.

`addDirectoryWatcher$(path, watcher)`

Registers `watcher`, a Callback (see DTCallback) to be called when an operation is performed on the directory specified by `path`. This function is provided to allow graphic representations of directories to be updated as actions occur in the underlying directories. Only actions that occur in the named directory are reported; actions in subdirectories are not.

Returns `{path, success, [error], [key]}`, where the key returned is necessary to remove a directory watcher with `removeDirectoryWatcher()`. Removing directory watchers is encouraged when they are no longer needed, in the interest of efficiency.

Possible errors include:, ERR_BADPATH, ERR_NOPATH, and ERR_NOTDIR.

The Callback provided is called with an object that always has an `action` property that names the action being performed, and a `path` property, which provides the path to the directory being watched. Additional properties provide details on the action.

The following `action` / parameter combinations are supported:

`ANCESTOR_RENAME` / <none>

an ancestor directory has been renamed.

`DIR_DELETE` / <none>

the watched directory has been removed. This watcher will never be called again, even if the directory is recreated.

`DIR_MODIFY` / <properties>

the watched directory has been modified. <properties> are one or more of the properties given by `DIR_ADD` that have changed: either `shared`, or `iconPathType` and `iconPath` (in the latter case, both may be `null`; which indicates icon removal).

`DIR_RENAME` / `oldName`, `newName`

the directory has been renamed.

`DIR_UPDATE` / `entries`

The directory's contents have been updated from the server. There may be no changes, a single change, or multiple changes; it is best to rebuild any views of the watched directory. `entries` is an array of the objects returned by `getEntry$()`.

`ENTRY_ADD` / <properties>

a new entry has been added, described by <properties>.

`ENTRY_DELETE` / `name`

the entry specified by `name` has been deleted.

ENTRY_MODIFY / name, <properties>

the entry has been modified. <properties> are one or more of the properties given by ENTRY_ADD that have changed: either shared, or iconPathType and iconPath (in the latter case, both may be null; which indicates icon removal).

ENTRY_RENAME / oldName, newName

the entry has been renamed.

Note that the watcher will be called immediately with an action of DIR_INIT, before addDirectoryWatcher$() returns. This provides an initial view of the directory's contents so that the watcher is guaranteed to be in sync. An entries property is returned with DIR_INIT as with DIR_UPDATE.

The following list serves as a summary of when and how watchers are called:

makeDirectory$(): calls ENTRY_ADD for the directory containing the new directory.

move$() to a different directory: calls ENTRY_DELETE for the original containing directory; DIR_DELETE for the moved node if a directory, and recursively for subdirectories; and ENTRY_ADD for new containing directory.

move$() to the same directory (i.e., rename): calls ENTRY_RENAME for the containing directory; DIR_RENAME for the moved node, if a directory; and ANCESTOR_RENAME recursively for subdirectories of the moved node, if a directory.

put$(): calls ENTRY_ADD for the directory containing the new file.

removeDirectory(): calls ENTRY_DELETE for the containing directory(ies); and DIR_DELETE for the removed directory.

setIcon$(): calls ENTRY_MODIFY for the containing directory(ies); and DIR_MODIFY for the moved node, if a directory.

smartlink$(): calls ENTRY_ADD for the directory containing the new smartlink.

symlink$(): calls ENTRY_ADD for the directory containing the new symlink.

unlink$(): calls ENTRY_DELETE for the containing directory(ies).

Watchers are always called in the order indicated. When calls are made recursively for subdirectories, a preorder depth-first traversal is used, but the order of calls among sibling subdirectories is undefined. The order of calls among multiple containing directories is also undefined.

If there are multiple events that cause watchers to be called, the watcher calls from the various events will always occur in the same order as the events themselves.

basename(path)

Returns the last item in path. (For example: path = "path/to/file," basename = "file.") An empty string indicates that the path is the root directory.

`canonicalize(path, default_dir)`

Turns a path into a canonical path (an unambiguous, simplified form), with `path` specifying a path, relative path or filename, and `default_dir` specifying an optional directory to use when the path is relative.

`dirname(path)`

Returns the directory part of `path`. (For example: path = "path/to/file," dirname = "path/to.") An empty string indicates the path is either the root directory or something in it.

`get$(path)`

Retrieves a RootPersistentObject from the filesystem. Path may be within the current user's filesystem, or to another user's filesystem.

Returns `{path, success, [error], [object]}`

Possible errors include: `ERR_BADCHARS`, `ERR_NOPATH`, `ERR_NOTDIR`, `ERR_FILENOTFOUND`, and `ERR_NOTFILE`.

`getDirectoryEntryArray$(path)`

Returns information about the entries in a given directory. Path may be within the current user's filesystem, or to another user's filesystem.

Returns `{path, success, [error], [entries]}`, where `entries` is an array of the objects returned by `getEntry$()`.

Possible errors include: `ERR_BADCHARS`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`getDirectoryEntries$(path)`

Returns information about the entries in a given directory. Path may be within the current user's filesystem, or to another user's filesystem.

Returns `{path, success, [error], [entries]}`, where `entries` is a hash of the objects returned by `getEntry$()`. (A hash is an object with properties whose names denote the names of the entries, and whose values are object references.)

Possible errors include: `ERR_BADCHARS`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`getEntry$(path)`

Returns information about a given directory entry. Path may be within the current user's filesystem, or to another user's filesystem.

Returns `{path, success, [error], [entry]}`, where `entry` is an object with the properties:

`name`: the name of the entry within the directory specified.

`shared`: a Boolean indicating whether the entry is directly shared. Indirect sharing (sharing of an ancestor directory) is not indicated. This property is not present for entries in other users' spaces.

`iconPathType`, `iconPath`: if an icon has been established with `setIcon$()`, these properties are present and describe the icon.

`rpoType`: a string that gives the class name of the RPO that the file represents (if any).

`dateCreated`: a Date object specifying when the entry was created.

`symlinkPath`: gives the path stored by a symlink. Present only for symlink entries.

Possible errors include: `ERR_BADPATH`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`getErrorString(err)`

Converts an error number returned by a FileSystem function into an error message string that may be displayed to users.

`getLinks$(target)`

Retrieves a list of the traceable links to a node specified by `target`. Traceable links include hard links and smart links.

Returns `{target, success, [error], [HARD], [SHARE], [USER]}` where `HARD`, `SHARE` and `USER` are each arrays of paths. `target` may be a path string or a reference to an RPO. If `target` specifies an RPO that is not in the filesystem, `getLinks$()` will return `{target:target, success:true}`. (Note the absence of the `HARD` property in the return value.)

Possible errors include: `ERR_BADARGS`, `ERR_BADCHARS`, `ERR_BADPATH`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`getSmartLinkTarget$(path)`

Retrieves the path to the target of the smart link with the specified `path`.

Returns `{path, success, [error], [targetPath]}`

Possible errors include: `ERR_BADCHARS`, `ERR_BADPATH`, `ERR_NOPATH`, `ERR_NOTDIR`, and `ERR_NOTSMARTLINK`.

`makeDirectory$(path)`

Makes a new directory at `path`, and generates all parent directories, if necessary.

Returns `{path, success, [error], [object]}`

Possible errors include: `ERR_BADCHARS`, `ERR_BADPATH`, and `ERR_EXISTS`.

`makePrivateURL(path)`

Constructs an URL to a file in the current user's filesystem, where `path` is the full path to the desired file. This URL will work only within the Desktop.com site, and only for the current user. The target file must be an uploaded file (RawFile RPO type). Returns an URL string.

`move$(origPath, newPath)`

Moves the file or directory at `origPath` to `newPath.` If `newPath` is not absolute, it will be assumed to be rooted in the same directory as the node being moved. If the node has smart links other than through the specified `origPath`, they are unaffected. To determine if other links exist, call `getLinks$()`. Symlinks referencing `origPath` will break without warning.

Note that the directory containing the desired target path must exist: `makeDirectory$()` is not called automatically to make parents of `newPath`.

Returns `{origPath, newPath, success, [error], [whichPath]},` where `whichPath` will be either "`ORIG`" or "`NEW`" to indicate the path returning the error.

Possible errors include: `ERR_BADARGS`, `ERR_FILENOTFOUND` (there is nothing at `origPath`), `ERR_EXISTS`, `ERR_BADCHARS`, `ERR_BADPATH`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`put$(path, obj)`

Links an RPO into the filesystem. `obj` is a reference to the RPO to be linked.

Returns `{path, obj, success, [error]}`

Possible errors include: `ERR_BADCHARS`, `ERR_BADPATH`, and `ERR_EXISTS`.

`putTemp$(path, prefix, obj)`

Links an RPO into the filesystem, and generates a filename guaranteed to be unique. `path` specifies a directory; `prefix` (which may be an empty string) specifies initial characters for the filename.

Returns `{obj, path, success, [error]}`, where `path` is the full path.

Possible errors include: `ERR_BADCHARS`, `ERR_BADPATH`, `ERR_NOPATH`, and `ERR_NOTDIR`.

`removeDirectoryWatcher$(key)`

Removes a watcher established with `addDirectoryWatcher$()`. The `key` returned by `addDirectoryWatcher$()` must be provided.

`removeDirectory$(path)`

Removes the empty directory at `path`. To remove a directory that is not empty, first `unlink$()` everything in it, then call `removeDirectory$()`.

Returns `{path, success, [error]}`

Possible errors include: ERR_BADARGS, ERR_BADCHARS, ERR_NOPATH, ERR_NOTDIR, and ERR_NOTEMPTY.

`setIcon$(nodePath, iconPathType, iconPath)`

Establishes an icon for the node defined by `nodePath`. `iconPathType` indicates the type of path to which `iconPath` points. If `iconPath` is "URL", then `iconPathType` is an URL to an image.

Passing `null` for `iconPathType` causes any icon at the node to be removed.

Returns `{nodePath, iconPathType, iconPath, success, [error]}`

Possible errors include: ERR_BADARGS, ERR_BADCHARS, ERR_BADPATH, ERR_NOPATH (`nodePath` does not exist), and ERR_NOTDIR (`nodePath` attempts to traverse a non-directory).

`smartlink$(source, target)`

Creates a smart link from `source` to `target`, both of which are path strings.

Returns `{source, target, success, [error]}`

Possible errors include: ERR_BADARGS, ERR_BADCHARS, ERR_BADPATH, ERR_NOPATH, ERR_NOTDIR, ERR_EXISTS, and ERR_NOSMART.

`symlink$(source, target)`

Creates a symlink at `source` which points to `target`, both of which are path strings.

Returns `{source, target, success, [error]}`

Possible errors include: ERR_BADARGS, ERR_BADCHARS, ERR_NOPATH, ERR_NOTDIR, and ERR_EXISTS.

`unlink$(path)`

Removes a link to a node. If a hard link is removed, and the node has any smart links, the smart links are also removed.

Returns `{path, success, [error]}`

Possible errors include: ERR_BADARGS, ERR_BADCHARS, ERR_NOPATH, ERR_NOTDIR, and ERR_DIRECTORY.

`updateDirectory$(path)`

Updates the entries for the directory defined by `path` from the server. Call `updateDirectory$()` only if another user may have modified the contents of a directory.

`validate(path)`

Returns `false` if `path` contains unacceptable characters; `true` if `path` is valid.

Errors Returned:

ERR_BADARGS: invalid argument(s).

ERR_BADCHARS: illegal characters in path.

ERR_BADPATH: syntax error in a specified path.

ERR_DIRECTORY: the path specifies a directory. Use `removeDirectory()` instead.

ERR_EXISTS: there is already something at the specified source path.

ERR_FILENOTFOUND: the specified directory exists, but the specified filename within that directory does not.

ERR_NOPATH: the directory containing the specified source path does not exist.

ERR_NOSMART: the target of a smartlink can only be a file or directory. A target path was specified that points at something else.

ERR_NOTDIR: the specified source path attempts to traverse a non-directory.

ERR_NOTEMPTY: the specified directory is not empty.

ERR_NOTFILE: the specified path points at something other than a file.

ERR_NOTSMARTLINK: the path points to something that is not a smart link.

DTFileUpload

The DTFileUpload package allows users to upload files from their computers to the Desktop server.

Classes

None.

Functions

`upload$(targetDirPath)`

Displays a dialog in which the user selects one or more files to upload from their computer. `targetDirPath` is a path string that specifies a directory in the user's Desktop.com filesystem. The uploaded files will be placed in this directory, with names that match the original names of the uploaded files as closely as possible.

Returns after the user closes the dialog. Returns `{success, path, files}`. `success` is normally `true`; if `false`, an error occurred. `path` is the `targetDirPath` specified, and `files` is an array of file name strings for the uploaded files inserted into the user's Desktop.com filesystem. Note that the length of the `files` array may be zero, indicating that the user canceled the upload operation.

DTFlowLayoutManager

The DTFlowLayoutManager package defines the FlowLayoutManager class.

Classes

FlowLayoutManager

Arranges Components from left to right in wrapping rows. Line breaks are determined automatically, but may also be manually set.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`FlowLayoutManager()`

Methods

getHorizontalPadding()	setHorizontalPadding(p)
getVerticalPadding()	setVerticalPadding(p)
lineBreak(item)	

`getHorizontalPadding()`
: Returns the horizontal padding in pixels.

`getVerticalPadding()`
: Returns the vertical padding in pixels.

`lineBreak(item)`
: Adds a line break after a Component: either the Component with (zero-based) index number `item`, or the last Component added if `item` is not specified.

`setHorizontalPadding(p)`
: Sets the horizontal padding in pixels.

`setVerticalPadding(p)`
: Sets the vertical padding in pixels.

Functions

None.

DTFontProber

The DTFontProber package defines the FontMetrics class.

Classes

FontMetrics

Encapsulates accurate measurements of various display size properties for a single font and font size.

In addition to raw text metrics, information is available on the sizing of text-input HTML form elements in the given font and font size. The two kinds of form elements are text fields (single-line, used in DTTextInputField components) and text boxes (multi-line, used in DTNativeTextInputBox components).

See also DTTextInputField, DTNativeTextInputBox, and DTStyle.

Inherits from DTObjectFramework.DTObject.

Constructor

None. (The FontMetrics class should not be instantiated directly. `ProbeFont$()` will return an instance of it.)

Methods

charCodeWidth(c)	textboxRows(height)
charHeight()	textboxHeight(rows)
charWidth(c)	textboxWidth(cols)
getStringOfWidth(str, width)	textfieldHeight()
stringWidth(s)	textfieldSize(width)
textboxCols(width)	textfieldWidth(size)

`charCodeWidth(c)`

Returns the pixel width in this font of the character specified by `c`, an integer representing an ASCII character code.

`charHeight()`

Returns the pixel height of the font.

`charWidth(c)`

Returns the pixel width in this font of the first character in the string specified by `c`.

`getStringOfWidth(str, width)`

Returns the largest left substring of the string `str` with a pixel width in this font less than or equal to `width`.

`stringWidth(s)`
: Returns the pixel width of the string `s`.

`textboxCols(width)`
: Returns the maximum number of columns a text box using this font and font size may have before its pixel width exceeds `width`.

`textboxRows(height)`
: Returns the maximum number of rows a text box using this font and font size may have before its pixel height exceeds `height`.

`textboxHeight(rows)`
: Returns the pixel height of a text box using this font and font size and having the number of rows specified by `rows`.

`textboxWidth(cols)`
: Returns the pixel width of a text box using this font and font size and having the number of columns specified by `cols`.

`textfieldHeight()`
: Returns the pixel height of a text field using this font and font size.

`textfieldSize(width)`
: Returns the maximum size a text field using this font and font size may have before its pixel width exceeds `width`.

`textfieldWidth(size)`
: Returns the pixel width of a text field using this font and font size, and having the size specified by `size`.

Functions

`ProbeFont$(font_name, font_size)`
: Creates and returns an instance of FontMetrics for the specified `font_name` and `font_size`. Both parameters are strings, and both are optional. Omitting either parameter will yield metrics that use the browser default for the omitted parameter.

  `font_name` may take any value that is valid for the `font-family` attribute of a CSS style. Recognized values include `serif`, `sans-serif`, `cursive`, `fantasy`, and `monospace`. `font-size` may take any value that is valid for the `font-size` attribute of a CSS style. Recognized values include absolute sizes (`xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`), relative sizes (`smaller`, `larger`), font lengths (any positive integer followed by `pt` or `em`), and relative percentages (any positive integer followed by %).

CSS styles are encapsulated by the DTStyle.Style class, which allows fonts and font sizes (and other attributes) to be specified for Components.

Note that the default fonts that browsers use for raw text, text fields, and text boxes differ from one another. When `font_name` and `font_size` are specified, these three text types will use the same font.

DTGridLayoutConstraints

The DTGridLayoutConstraints package defines the GridLayoutConstraints class.

Classes

GridLayoutConstraints

Defines the parameters used to construct a grid layout.

Constructor

`GridLayoutConstraints()`

Actions

None.

Methods

getAnchor()	setAnchor(a)
getBreak()	setBreak(b)
getFill()	setDefaults()
getSpanX()	setFill(fill)
getSpanY()	setSpanX(s)
getWeightX()	setSpanY(s)
getWeightY()	setWeightX(w)
lineBreak()	setWeightY(w)

`getAnchor()`
: Returns the anchor which positions the component within its cell(s).

`getBreak()`
: Returns the location for a break.

`getFill()`
: Returns the fill style, which defines how the component should be resized by the layout manager in order to fill the space that it is in.

`getSpanX()`
: Returns the horizontal span (the number of columns that the component crosses along the X axis).

`getSpanY()`
: Returns the vertical span (the number of columns that the component crosses along the Y axis).

`getWeightX()`
: Returns the horizontal (along the X axis) weight of the cell.

`getWeightY()`
: Returns the vertical (along the Y axis) weight of the cell.

`lineBreak()`
: Inserts a line break.

`setAnchor(a)`
: Sets the anchor to position the component within its cell(s). Valid input includes: GridLayoutConstraints.NORTH, GridLayoutConstraints.NORTH_EAST, GridLayoutConstraints.EAST, GridLayoutConstraints.SOUTH_EAST, GridLayoutConstraints.SOUTH, GridLayoutConstraints.SOUTH_WEST, GridLayoutConstraints.WEST, GridLayoutConstraints.NORTH_WEST, and GridLayoutConstraints.CENTER (default).

`setBreak(b)`
: Sets the location for a break.

`setDefaults()`
: Resets all grid constraints to their default values.

`setFill(fill)`
: Sets the fill style, which defines how the component should be resized by the layout manager in order to fill the space that it is in. Valid input includes: GridLayoutConstraints.FILL_HORIZONTAL, GridLayoutConstraints.FILL_VERTICAL, GridLayoutConstraints.FILL_BOTH, and GridLayoutConstraints.FILL_NONE (default).

`setSpanX(s)`
: Sets the horizontal span (the span along the X axis). Span defines the number of columns that the components may cross, and must be a positive integer.

`setSpanY(s)`
: Sets the vertical span (the span along the Y axis). Span defines the number of columns that the components may cross, and must be a positive integer.

`setWeightX(w)`
: Sets the horizontal weight (the weight along the X axis). Weight affects the size of the cell in relation to the other cells in its row, and must be a number between 0 and 1.

`setWeightY(w)`
: Sets the vertical weight (the weight along the Y axis). Weight affects the size of the cell in relation to the other cells in its row, and must be a number between 0 and 1.

Functions

None.

DTGridLayoutManager

The DTGridLayoutManager package defines the GridLayoutManager class

Classes

GridLayoutManager

The layout manager for grid components.

Grids differ from tables in that grids are more rigid. All columns must be the same width, and all rows the same height across the grid.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`GridLayoutManager()`

Methods

`addComponent(component, constraints)`
: Adds a component with the specified constraints.

`layoutComponents()`
: Lays out the components for the grid.

Functions

None.

DTHelpWindow

The DTHelpWindow package defines a Web pane-based help window.

Classes

None.

Functions

`HelpWindow(parentApp, title, helpURL)`

Defines a new Help window as a child window of the parent application, using the parent application, the title for the new window, and the URL for the (initial) help page. On Internet Explorer, the Help window consists of a web pane inside an application window, with supporting navigation buttons. On Netscape Navigator, the Help window opens as a new browser window (outside of the Desktop.com environment).

DTHTMLBox

The DTHTMLBox package defines the HTMLBox class.

Classes

HTMLBox

Defines a pane with dynamic (pseudo) HTML content. Content is distinguished from pure HTML in that links may be set which activate programmatic events within an application. The links are specified as <a name="name" action="action">. When the link is clicked, the action listener is called with the "action" as a parameter.

The links specified by *name* and *action* must be created and defined by the developer. The action they specify is then passed to action listeners when the links are clicked.

Inherits from DTComponent.Component.

Constructor

`HTMLBox(html)`

`html` sets the HTML source file to be shown in the box

Actions

`link`: an action was generated after a link was clicked.

`mouse`: an action was generated due to a mouse event.

`other`: an action was generated for some other reason.

Methods

getActionType() setHTML(html)
getHTML()

`getActionType()`

Returns the type of action over a link. Possible values includes `link`, `mouse`, and `other`. This method should be called only after receiving an action from the HTMLBox.

`getHTML()`

Returns the HTML file for the frame.

`setHTML(html)`

Sets the HTML file for the frame.

Functions

None.

DTHTMLBrowser

The DTHTML package defines the HTMLBrowser class.

Classes

HTMLBrowser

Opens a new browser window within the Desktop.com environment, which allow users to surf pages consisting of panes or components. Links within the pages require action listeners to trigger browser updates.

Inherits from DTPane.Pane.

Constructor

`HTMLBox(html)`

Actions

None.

Methods

addPage(comp)	home()
back()	setSize(width, height)
forward()	

`addPage(comp)`
: Adds and displays a new page to the HTMLBrowser. The page may consist of panes or components.

`back()`
: Displays the previous page in the queue. If no such page exists, no action is taken.

`forward()`
: Displays the next page in the queue. If no such page exists, no action is taken.

`home()`
: Displays the home page, the first page the HTML browser receives using the `addPage()` method, in the browser.

`setSize(width, height)`
: Sets the width and height for the browser window in pixels.

Functions

None.

DTIconsView

The DTIconsView package defines the IconsView class.

Classes

IconsView

Defines a scroll pane in which icons composed of an image and associated text may be placed and arranged by the user. The window in which these icons are placed has a scrollbar on both its right and bottom sides. Users may drag and arrange icons anywhere within the scroll pane.

Inherits from DTScrollPane.ScrollPane.

Constructor

`IconsView(lm)`

`lm` is the layout manager to be associated with the component.

Actions

None.

Methods

addNode(node, width, height, align, tooltip)
removeNode()
enableDrag()
setColor()
moveIcon(event)
setSelectedNode(node)

`addNode(node, width, height, align, tooltip)`

Adds an icon to the pane, defined by its `width`; `height`; tooltip text; and text alignment, which may be `center`, `left`, or `right`. The icon is defined using `node.icon`; the text is `node.data.text`.

`clear()`

Removes all components from the pane except for the focus indicator.

`enableDrag()`

Enables dragging of icons within the pane: `true` if draggable; `false` if not.

`moveIcon(event)`

Sets an icon to move in response to the defined event.

`removeNode()`

Removes the specified node. Generates `node_removed` action for all action listeners.

`setColor()`

Sets the color for the text displayed in the icon, defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setSelectedNode(node)`

Sets the node that should be selected. Generates a `selected` action for the action listeners if the node is not `null`.

Functions

None.

DTImageArea

The DTImageArea package defines the ImageArea class.

Classes

ImageArea

Defines a component in which images may be placed. Three display modes are supported: `plain`, which displays the image in its absolute size, with no margins; `centered`, which displays the image in its absolute size, but allows the ImageArea component's size to be set, leaving any difference as a margin; and `stretch`, which forces the image dimensions to the specified size.

Inherits from DTComponent.Component.

Constructor

`ImageArea(imagePath, toolTip, mode)`

> `imagePath` defines the path to the image to be displayed; `toolTip` sets the ALT text to be displayed during mouse-over events; and `mode` sets the display mode: `PLAIN`, `CENTER`, or `STRETCH`.

Actions

`clicked`: the icon has been clicked (including clicks in margins when in centered mode).

Methods

getBorder()
getImage()
getMode()
getToolTip()
setBorder(thickness)
setImage(imagePath)
setMode(newMode)
setSize(w, h)
setToolTip(toolTip)

`getBorder()`

> Returns the border thickness in pixels.

`getImage()`

> Returns the absolute path string for the image.

`getMode()`

> Returns the display mode.

`getToolTip()`

> Returns the text that appears when the mouse is over the image (the ALT text).

`setBorder()`
: Sets the border thickness in pixels. Default is 0.

`setImage(imagePath)`
: Sets the absolute path string for the image.

`setMode(newMode)`
: Sets the display mode. Possible values include `plain` (default), `centered`, and `stretch`.

`setSize(w, h)`
: Sets the width and height for the image area. `setSize()` has no immediate effect in plain mode, since plain mode images are naturally sized, but will be used for all subsequent `setMode()` calls. If an image area is resized to a smaller size, be certain to repaint any parent objects behind it in the window.

`setToolTip(toolTip)`
: Sets the text string that appears when the mouse is over the image (the ALT text).

Functions

None.

DTImageButton

The DTImageButton package defines the DTImageButton class.

Classes

ImageButton

Defines a button Component, which may include up to three images: one each for inactive, mouseover, and mouse clicked events.

Inherits from DTComponent.Component in Netscape Navigator; and DTImageArea.ImageArea on Internet Explorer.

Constructor

`ImageButton(images)`

> `images` is an array of images. The images array is indexed as `[inactive, mouseover, mousedown]`. If the mouseover or mousedown elements are undefined, the inactive image substitutes for them.

Actions

None.

Methods

handleMouseEvent(e)	setImage(image)
setClicked(isClicked)	setImages(images)

`handleMouseEvent(e)`

> Called when a mouse event is received.

`setClicked(isClicked)`

> If `isClicked` is `true`, sets the image button to its defined `mousedown` appearance; if `false`, sets it to its `inactive` appearance.

`setImage(image)`

> Sets the default image to be used for the button. `image` is an URL string, as for the DTButton constructor.

`setImages(images)`

> Sets the images to be used for the button. `images` is an array of URL strings, with the URL format as for the DTButton constructor, indexed as `[inactive, mouseover, mousedown]`.

Functions

None.

DTImageResize

The DTImageResize package defines an interface to a server-side utility that creates thumbnails of uploaded image files.

Classes

None.

Functions

`resize$(source_path, dest_path, x_max, y_max)`

Creates a thumbnail image. `source_path` and `dest_path` are path strings that specify the locations of the source and destination files, respectively, in the current user's filesystem. The source file must be an uploaded GIF or JPEG image file, and there must not be a anything located at the destination path. `x_max` and `y_max` are positive integers that specify the maximum dimensions for the thumbnail. The thumbnail will be of the same file type as the source file.

Returns an object with named properties. If the property `success` is `true`, the operation succeeded and the remaining properties are meaningful; if not, the operation failed. The remaining properties `x_source` and `y_source` define the pixel size of the source image, and `x_dest` and `y_dest` define the pixel size of the thumbnail.

DTIncrSlider

The DTIncrSlider package defines an incremental slider widget, as distinct from a continuous slider, which DTSlider (the parent class) provides.

Classes

IncrSlider

Defines an incremental slider widget.

Incremental sliders take on discrete values of the form (min + n * incr), where n takes on positive integer values and incr is an increment value specified by the widget's calling code.

Dragging and bumping behaviors are different for an incremental slider than for a continuous slider. Dragging jumps among the allowable quantified values, rather than sliding smoothly; and bumping jumps by increments, rather than by a fixed fraction of the slider's total travel.

Incremental sliders are displayed with tick marks to indicate the allowed slider-button center positions. Labels may be provided either for the first and last ticks or for all ticks. Ticks and labels appear on the bottom side of a horizontal slider, and on the right side of a vertical slider.

Either `setNumIncrements()` or `setIncrementSize()` must be called during initialization of an incremental slider. If both are called, the second call is used and the first ignored. There must be a positive integer number of increments, and increment size must exactly divide the difference between min and max into equal segments.

Increment and tick-label setup is immutable.

When `setSize()` is called on a horizontal incremental slider with tick labels, the width argument will be used to size and position all sub-components under the assumption that the tick label text at the two endpoints does not extend beyond the boundary of the slider's background pane.

Note that long text strings may cause the labels to be truncated. To prevent this, set the slider's `lengthPadding` property to the number of extra pixels required to accommodate the labels. `lengthPadding` specifies the total number of extra pixels, not the number of pixels per end (half of the total). `lengthPadding` must be set to an even number or incorrect behavior will result.

Inherits from DTSlider.Slider.

Constructor
`IncrSlider()`

Actions
None.

Methods

getIncrementSize()	setNumIncrements(n)
getNumIncrements()	setTickLabels(labels)
setIncrementSize(incrSize)	setValue(value)

`getIncrementSize()`
: Returns the size of the increment in pixels.

`getNumIncrements()`
: Returns the number of increments for the slider bar.

`setIncrementSize(incrSize)`
: Sets the size of the increment in pixels.

`setNumIncrements(n)`
: Sets the number of increments for the slider bar. Note that `n` specifies the number of segments, not the number of possible values (ticks), which is one greater than the number of segments.

`setTickLabels(labels)`
: Sets the labels to be used for the tick marks. `labels` should be an array of strings. If the length of this array is 2, the two strings will be used for the lowest- and highest-valued ticks. If the length is equal to `getNumIncrements()` + 1, the strings will be used successively for all ticks. An array of any other length will be ignored.

`setValue(value)`
: The argument to `setValue()` will be silently rounded to the nearest allowable value. Call `getValue()` to determine the effects of rounding.

Functions

None.

DTKeyEvent

The DTKeyEvent package defines the KeyEvent class.

Classes

KeyEvent

The DTKeyEvent class defines a list of constants which allow you to grab almost every available keyboard input as an event.

Constructor

`KeyEvent()`

Constants

KeyEvent constants are defined using the syntax `KeyEvent.kVK_`*`key`*. Such as:

```
KeyEvent.kVK_2
KeyEvent.kVK_S
KeyEvent.kVK_PERIOD
```

For a complete list of defined constants, see Appendix II: KeyEvent Constants.

Functions

None.

DTLabelledTextBox

The DTLabelledTextBox package defines the LabelledTextBox class.

Classes

LabelledTextBox

Defines a window object which contains a label and a text input field. The developer-defined label is placed to the left of the text input field, which may contain text when the box is first displayed.

Inherits from DTPane.Pane.

Constructor

`LabelledTextBox(label, entry_size)`

> `label` defines the text to be used for the label; and `entry_size` defines the size of the TextInputField in characters.

Actions

None.

Methods

getText()
getTextInputField()
getTextLabel()
makeBold()
setBackgroundColor(color)
setText(text)
setTextColor(color)

`getText()`

> Returns the text from the user input field.

`getTextInputField()`

> Returns the `TextInputField` object.

`getTextLabel()`

> Returns the `TextLabel` object.

`makeBold()`

> Makes the label text bold

`setBackgroundColor(color)`

> Sets the background color for the box. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setText(text)`

> Sets the text for the user input field.

`setTextColor(color)`

Sets the text color for the box. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

Functions

None.

DTLayoutManager

The DTLayoutManager package defines the abstract base class for Desktop.com layout managers. All layout managers are a subclass of the LayoutManager class.

Classes

LayoutManager

Defines a layout manager: an object associated with a Container (see DTContainer.Container), which controls the size and/or position of the Container and/or its children.

To use a layout manager, instantiate the appropriate LayoutManager subclass and pass the resulting object to either the constructor or the `setLayoutManager()` method of the relevant Container. Be certain to instantiate a separate layout manager object for each container, and do not share layout managers between containers.

Some layout managers function automatically, while others require configuration, either before or during the process of adding Components to the associated Container. LayoutManager subclass methods enable layout manager configuration.

Most layout managers' behavior is affected by the order in which Components are added to the associated Container. Once components have been added to a container, there is no way to change their order other than to remove and then replace all Components in the desired order. After all Components have been added to the layout manager's associated Container, layout will be performed each time the Container is resized. Layouts may also be forced by calling Container.doLayout().

If you are using a layout manager and seeing strange results, try calling the `setCautious()` method of LayoutManager. By default, layout managers operate in non-cautious mode; `setCautious(true)` will force them into cautious mode, which is more conservative but slightly less efficient.

To implement a new kind of layout manager, either generic or single-purpose, create a subclass of LayoutManager and implement a single method in that class called `layoutComponents()`. This method will be called each time layout is to be performed, such as when the layout manager's associated Container is resized, or an explicit call is made to `Container.doLayout()`. A layout manager must contain the two class-default properties NEED_CONTAINER_SIZE and NEED_COMPONENT_SIZE. (To set a class-default property, use DTObjectFramework.SetDefault().) If the layout manager requires a defined size for the associated Container before `layoutComponents()` will work correctly (i.e., if `layoutComponents()` calls `this.container.getSize()`), set NEED_CONTAINER_SIZE to `true`; otherwise set it to `false`. If `layoutComponents()` needs a defined size for any child Components, set NEED_COMPONENT_SIZE to `true`; otherwise set it to `false`.

Within `layoutComponents()`, you may query and operate upon the *container* and *items* properties of the *this* object. `container` is a reference to the associated Container object, and items is an array containing a reference to each Component in the associated Container.

If a layout manager subclass seems always to need a call to `setCautious(true)` to make it work correctly, set the class-default property `cautious` to `true`.

Inherits from DTObjectFramework.DTObject.

Constructor

`LayoutManager()`

Methods

`setCautious(cautious)`

> Sets the layout mode to cautious. If `true`, forces the layout manager into cautious mode; if `false`, into non-cautious mode. Cautious mode draws the components before arranging them; non-cautious arranges, then draws the components, which is faster, but may engender visual mistakes.

Functions

None.

DTLinkArea

The DTLinkArea package defines the LinkArea class.

Classes

LinkArea

Defines a component to contain the hypertext link.

Inherits from DTTextBox.TextBox.

Constructor

`LinkArea(text, alignment)`

Sets the `text` to be linked, and its `alignment` within the component.

Actions

None.

Methods

`setCb(cb)`

Sets the callback, which is called when the link is clicked.

Functions

None.

DTListBox

The DTListBox package defines the ListBox class.

Classes

ListBox

A scrolling list of text items.

Inherits from DTScrollPane.ScrollPane.

Constructor

`ListBox()`

Actions

None.

Methods

appendListItem(text, data)	removeListItem(index)
clearListItems()	setItemData(index, data)
getItemData(index)	setItemText(index, text)
getItemText(index)	setSelectedData(data)
getSelectedData()	setSelectedItem(index)
getSelectedItem()	setSelectedText(text)
getSelectedText()	

`appendListItem(text, data)`

Adds a `text` item to the end of the list. `data` is an optional data element that may be stored with the list item. `Data` will be returned to be used when events are fired.

`clearListItems()`

Removes all the items from the list.

`getItemData(index)`

Returns the data associated with the item defined by `index`, or `null` if that index item does not exist.

`getItemText(index)`

Returns the text associated with the item defined by `index`, or `null` if that index item does not exist.

`getSelectedData()`

Returns the data associated with the item that is currently selected, or `null` if nothing is selected.

`getSelectedItem()`
: Returns the index number of the item that is currently selected, or `null` if nothing is selected.

`setSelectedText(text)`
: Returns the text of the item that is currently selected, or `null` if nothing is selected.

`removeListItem(index)`
: Removes the item specified by `index` from the list.

`setItemData(index, data)`
: Sets the data to be associated with the item defined by `index`, or `null` if that index item does not exist.

`setItemText(index, text)`
: Sets the text to be associated with the item defined by `index`, or `null` if that index item does not exist.

`setSelectedData(data)`
: Sets the data to be associated with the item that is currently selected, or `null` if nothing is selected.

`setSelectedItem(index)`
: Sets the index number of the item that is currently selected, or `null` if nothing is selected.

`setSelectedText(text)`
: Sets the text of the item that is currently selected, or `null` if nothing is selected.

Functions

None.

DTMenu

The DTMenu package defines the Menu class.

Classes

Menu

Defines a pop-up or pull-down menu widget.

Menus are made up of three types of entries: *items*, selectable text entries; *submenus*, text entries that cause additional menus to cascade; and *separators*, horizontal markers that separate categories of entries. Items and sub-menus always display text, and may include an icon to the left of the text.

Menus may be created in two ways: by entry list and incrementally. To use an entry list, pass an array of entry definitions to the Menu constructor to define any number of items, separators, and sub-menus with a single call. To set up a menu incrementally, make one call to `addItem()`, `addSeparator()`, or `addSubmenu()` for each entry to be added. After menu construction, entries may be added or removed incrementally at any time. It is also possible to enable and disable individual entries at any time, and change the text or icon of any entry.

All entry manipulation is keyed by entry IDs, which must be supplied by callers when adding entries. IDs may be any string or number. They must never be `null`, as `null` is used to indicate that no item is selected. IDs must always be unique within a Menu and all its submenus, or incorrect behavior will result. The uniqueness of newly supplied IDs is checked in some circumstances, but not all.

When an item is selected in a menu or any of its submenus, the menu disappears and a `selected` action is emitted. There are two ways to listen for this action: with an action listener applied to the Menu as a whole, which calls `getSelectedItemId()` to determine what was selected; or by applying action listeners to individual items using `setItemListener()`.

Do not insert a Menu into a Container as a child component: Menus are automatically parented by the screen.

Inherits from DTContainer.Container.

Constructor

`Menu(app, entryList)`

`app` (optional) specifies the application to be associated with the menu.

`entryList` is also optional; if omitted, an empty menu is created. If given, it contains an ordered array of objects; each of which specifies an entry using the following named properties:

`id`*:* specifies the ID of the entry. Required.

`separator`: if present and `true`, the entry represents a separator, and all properties except `id` are ignored.

`menu`: if this property is present, the entry is a submenu, and this property provides a reference to the submenu definition, which follows the same format as entryList itself. If not present, the entry is an item.

`text`: specifies the text to be displayed for the item. Required for an item or submenu.

`icon`: if present, specifies the path of the icon to display to the left of the entry.

`enabled`: if present, specifies whether the entry is initially enabled or not: `true` if enabled; `false` if not. If absent, the entry will be enabled.

`listener`: if the entry is an item and this property is present, it provides the name of a listener function to be called when the item is selected. The application object established with the `app` argument, or with `setApp()`, will be assumed to contain a listener method by this name.

Actions

`selected`: an item has been selected by the user.

Methods

addItem(id, text, iconPath, listener, enabled, beforeId)
addSeparator(id, beforeId)
addSubMenu(id, text, menu, iconPath, enabled, beforeId)
advanceSelection(direction)
cascade(direction)
clearBottomAnchor()
getEntries()
getEntryText(id)
getMenu(id)
getParentage()
getSelectedItemId()
handleSelection()
isEntryEnabled(id)
reinit(entryList)
removeEntry(id)
setApp(app)
setBottomAnchor(left, bottom)
setEntryEnabled(id, enabled)
setEntryIcon(id, iconPath)
setEntryText(id, text)
setItemListener(id, listener)
setLazyEvaluator(func, alwaysCall)
setLocation()
setSelectionToEnd(whichEnd)
setVisible(visible)
undisplay()

Note that all methods that take an `id` parameter are recursive: they will find a specified ID anywhere in the top-level menu or in any submenus.

Note also that any method which changes the appearance of a menu, such as `setText()` or `addSeparator()`, will not take effect until the menu is hidden and redrawn.

`addItem(id, text, iconPath, listener, enabled, beforeId)`

Adds an item to the menu, with the given `id` and `text`. `iconPath` (optional) defines an icon to be placed to the left of the text entry. `listener` (optional) provides the name of an action listener to be associated with the menu item. `enabled` (optional) defines whether the submenu is enabled (default is `true`). `beforeId` (optional) specifies the ID of an existing entry before which the given entry is to be inserted. If `beforeId` is omitted, the entry is inserted at the end of the top level menu.

`addSeparator(id, beforeId)`

Adds a separator to the menu with the specified `id`. `beforeId` (optional) specifies the ID of an existing entry before which the given entry is to be inserted. If `beforeId` is omitted, the entry is inserted at the end of the top level menu.

`addSubMenu(id, text, menu, iconPath, enabled, beforeId)`

Adds a submenu entry with the given `text` and `id`. `menu` is a reference to the submenu to be associated with the entry. `iconPath` (optional) defines an icon to be placed to the left of the text entry. `enabled` (optional) defines whether the submenu is enabled. `beforeId` (optional) specifies the ID of an existing entry before which the given entry is to be inserted. If `beforeId` is omitted, the entry is inserted at the end of the top level menu.

`advanceSelection(direction)`

Advances the selection through the menu entries in the specified direction (`-1` for backwards; `1` for forwards). This method is circular, skips selectors, and does not skip disabled entries.

`cascade(direction)`

Opens or closes submenus as appropriate. `cascade(1)` opens a submenu if the selection is on a submenu entry and no submenu is visible yet. `cascade(-1)` closes the submenu if the selection is on a submenu entry and the submenu from that entry is visible. Calls to `cascade()` are recursive; `cascade()` will perform multiple levels of cascade display and undisplay.

Returns `true` if it did anything; `false` if no action was taken.

`clearBottomAnchor()`

Clears the bottom anchor designation of a menu. To change a Menu's anchoring from bottom to top, call this method, then call `setLocation()`.

`getEntries()`

Returns an array representing all the entries in the menu. Each element in the array is an object with the following properties:

`id`: the ID of the entry. Always present.

`type`: the type of entry. Valid values are Menu.ENTRY_ITEM, Menu.ENTRY_SEP, and Menu.ENTRY_SUB. Always present.

`text`: the display text for the entry. Present for items and submenus.

`icon`: the icon path for the entry. May be `null`. Present for items and submenus.

`enabled`: defines whether the entry is enabled. Present for items and submenus.

`menu`: a reference to the entry's submenu object. Present for submenus.

`getEntryText(id)`

Returns the text of the entry with the specified `id`.

`getMenu(id)`

Returns a reference to the menu associated with the entry that has the specified `id`.

`getParentage()`

Returns the position of a submenu within its parent menu, if one exists. Returns `null` if the menu is not a submenu. If it is a submenu, returns an object with the properties `menu`, a reference to the parent menu, and `id`, the ID of the parent Menu entry from which the Menu cascades.

`getSelectedItemId()`

Returns the selected item's ID. Call this method in response to the `selected` action.

`handleSelection()`

Causes the menu to behave as though it had received a mouse-up event. If an item is selected and enabled, the menu will disappear and emit the `selected` action.

`isEntryEnabled(id)`

Returns a Boolean indicating whether the entry with the specified `id` is enabled.

`reinit(entryList)`

Clears all entries from the Menu, and repopulates the Menu with the entries defined in `entryList` (which follows the same format as the `entryList` parameter to the Menu constructor).

`removeEntry(id)`

Removes the entry with the specified `id` from the menu.

`setApp(app)`

Sets the application to be associated with the menu. This is the application object that will be assumed to contain methods with the names specified in `setItemListener()`, and the listener properties in entry lists passed to the constructor and `reinit()`.

`setBottomAnchor(left, bottom)`

Sets bottom-left screen-relative pixel coordinates for a Menu.

`setEntryEnabled(id, enabled)`

Enables the entry with the specified `id` if `enabled` is `true`; disables the entry if `false`.

`setEntryIcon(id, iconPath)`

Sets an icon to be displayed to the left of the entry with the specified `id`. Icons should be exactly Menu.ICON_WIDTH by Menu.ICON_HEIGHT pixels.

`setEntryText(id, text)`

Sets the text of the entry with the specified `id`.

`setItemListener(id, listener)`

Associates the specified `listener` (the name of a method in the application specified by the `app` parameter to the constructor, or with `setApp()`) with the menu item that has the specified `id`.

`setLazyEvaluator(func, alwaysCall)`

Allows the Menu's entries to be determined dynamically by an evaluation function when the menu is about to be displayed. The evaluation function may be specified with a direct function reference; with two arguments, the first an object and the second a method name; or with a method-name string, in which case the method will be assumed to exist within the application object specified in the constructor or with `setApp()`. The evaluation function must provide two parameters: the first receives a reference to the menu object that the evaluation function should populate; the second receives a callback that must be called when the menu is complete. (Declare the evaluation function as a blocking "$" function.) When the evaluation function is called, the menu will be empty.

If `alwaysCall` is `true`, the evaluation function will be called each time the menu is about to be redisplayed (that is, made visible after a call to `undisplay()`). If `alwaysCall` is `false` or `null`, the content of the menu will be cached and reused without update at redisplay.

`setLocation()`

Sets the top-left screen-relative pixel coordinates for a Menu.

`setSelectionToEnd(whichEnd)`

Moves the selection to the end of the Menu. `whichEnd` is `-1` for the first entry, or `1` for the last. Note that entries begin at the top and proceed to the bottom, regardless of whether the anchoring component or point is at the top or the bottom of the menu.

`setVisible(visible)`

Called to show a Menu initially, or to toggle visibility. In most cases, `undisplay()` will provide better performance than `setVisible(false)`. However, if multiple menus descend from a single anchor component, such as a menu bar, call `setVisible(false)` to allow the user to move rapidly between menus.

`undisplay()`

Removes a menu's screen components. Called automatically when a selection occurs.

Functions

None.

DTMenuBar

The MenuBar class defines a menu bar object, which is a horizontal array of buttons, each of which serves to anchor a menu. (Menu bars are normally created as part of AppWindows using the `setMenuBar()` method.)

Classes

MenuBar

Encapsulates a menu bar component: a horizontal array of menu anchors, with an associated Menu component for each.

In a menu bar, both the menus and their entries are keyed by ID. It is important to ensure that all items in an menu bar have a unique ID. This uniqueness is not checked, and it is the caller's responsibility to ensure it. As with menus, IDs may take any string or numeric value. `null` is never a valid ID.

There are two ways to interact with a menu bar: Use `getMenu()` to retrieve a reference to a particular menu, then manipulate that menu as a whole; or use the passthrough mutators and accessors to manipulate menu entries based solely on IDs.

Menu bars automatically determine their own height and width. Calling `setSize()` has no effect on a MenuBar.

Many of the methods of MenuBar are passthrough methods, which call their equivalents in the Menu class. All passthrough methods are keyed by entry ID. The MenuBar searches all of its Menus for the specified ID, then calls the equivalent method on the Menu that contains the specified ID. (This is one reason why it is important to ensure that entry IDs in Menus are unique across all Menus in a MenuBar.) The passthrough methods are not documented in detail here: see the equivalent methods of DTMenu for details.

See DTMenu for more information.

Inherits from DTContainer.Container.

Constructor

`MenuBar(app, entryList)`

`app` is an optional argument that specifies the Application object with which the MenuBar is to be associated. The use of this parameter is equivalent to calling `setApp(app)`. This is only relevant when using "listener" attributes for menu items.

`entryList` is also optional; if absent or `null`, an empty menu bar is created. If present, `entryList` is an array of objects, in which each object specifies a menu, and the order of objects specifies the order of menus. Each object specifies its entry using the following named properties:

`id`: specifies the ID of the menu. Required.

`text`: specifies the text to be displayed with the menu anchor. Required.

`menu`: defines a menu. Follows the format of the entryList argument to the Menu() constructor. Required.

`enabled`: if present, specifies whether the menu is initially enabled or not: `true` if enabled; `false` if not. If absent, the menu will be enabled.

Actions

`selected`: a menu item has been clicked.

Note that there are three ways to listen for selected actions: by adding an action listener to a MenuBar; by adding an action listener to an individual menu; or by calling `setItemListener()`. When using more than one of these, be certain not to respond to the same action twice.

Methods

addMenu(id, text, menu, enabled, beforeId)
getMenu(id)
getSelectedItemId()
isEntryEnabled(id)
isMenuEnabled(id)
peekSize()
reinit(entryList)
removeMenu(id)
setApp(app)
setMenuEnabled(id, enabled)
setMenuText(id, text)

Passthrough Methods (see Menu for more complete information)

addItem(id, text, iconPath, enabled, beforeId)
addSeparator(id, beforeId)
addSubMenu(id, text, menu, iconPath, enabled, beforeId)
getEntryText(id)
isEntryEnabled(id)
removeEntry(id)
setEntryEnabled(id, enabled)
setEntryIcon(id, iconPath)
setEntryText(id, text)
setItemListener(id, listener)

`addMenu(id, text, menu, enabled, beforeId)`

Adds an anchor and associated `menu` to a MenuBar, with the specified `id` and `text`. (`id` must be unique across all menus and menu entries in the MenuBar.) `enabled` (optional) specifies whether the new anchor is to be enabled: `true` if enabled; `false` if not. If absent, the anchor will be enabled. `beforeId` (optional) specifies the ID of an anchor before which the new anchor is to be inserted. If absent, the new anchor will be placed at the end of the MenuBar.

`getMenu(id)`

Returns a reference to the Menu specified by `id`, or `null` if there is no such Menu.

`getSelectedItemId()`

Returns the ID of the selected item. Similar to `Menu.getSelectedItemId()`. Call this method in response to a `selected` event emitted from the MenuBar.

`peekSize()`

Returns an object `{w, h}` in which w and h indicate what the width and height of the MenuBar are or will be, in pixels, even if the MenuBar has yet to be painted. Be certain to add all Menus to a MenuBar before relying on the width reported by `peekSize()`.

`reinit(entryList)`

Resets and reinitializes a MenuBar's content with `entryList`: a menu bar definition that follows the same structure as the `entryList` parameter to the MenuBar constructor. (Similar to `Menu.reinit()`.)

`removeMenu(id)`

Removes the anchor specified by `id`, and its associated menu.

`setApp(app)`

Sets the Application object to be associated with a MenuBar. This method is relevant only when using "listener" attributes for menu items. (Similar to `Menu.setApp()`.)

`setMenuEnabled(id, enabled)`

Changes whether the anchor specified by `id` is enabled to respond to the mouse: `enabled` is `true` if enabled; `false` if not.

`setMenuText(id, text)`

Changes the text displayed anchor specified by `id`.

Functions

`isMenuEnabled(id)`

Returns whether the Menu specified by id is enabled: `true` if enabled; `false` if not enabled, or if no Menu with the specified `id` was found.

DTNamedCallback

The DTNamedCallback package defines a Callback with a unique name, by which it may be referenced. This package is useful if you cannot pass the callback object by reference, but you can pass a string. This package is seldom needed.

Classes

NamedCallback

Defines a callback.

Inherits from DTCallback.Callback.

Constructor

`NamedCallback(arg1, arg2, arg3)`

May be called in four different ways:

```
cb = new NamedCallback(func);
cb = new NamedCallback(func, data);
cb = new NamedCallback(obj, method_name);
cb = new NamedCallback(obj, method_name, data);
```

Methods

`getName()`

Returns the callback's name.

Functions

`Call(name, arg)`

Invokes a NamedCallback by name.

`Forget(cb)`

Unregisters the callback's name from the global NamedCallback namespace when it is no longer needed.

DTNativeComponent

The DTNativeComponent package defines the NativeComponent class.

Classes

NativeComponent

An abstract base class for DTAPI Components that are implemented using HTML form elements.

Constructor
None. Do not instantiate this class directly.

Actions
None.

Functions

None.

DTNativeTextInputBox

The DTNativeTextInputBox package defines the NativeTextInputBox class.

Classes

NativeTextInputBox

Defines a text input box, with a specified number of rows and columns into which users may type text entries, implemented using the browser's native HTML text area component.

Inherits from DTNativeComponent.NativeComponent.

Constructor

`NativeTextInputBox(text, cols, rows)`

`text` sets the text to appear in the box, beneath the specified number of column and row input fields.

Actions

None.

Methods

getText()	setSize(w, h)
setCols(cols)	setText(text)
setRows(rows)	setWrap(wrap)

`getText()`

Returns the text from the input box.

`setCols(cols)`

Sets the number of columns for the box.

`setRows(rows)`

Sets the number of rows for the box.

`setSize(w, h)`

Sets the width and height of the box in pixels.

`setText(text)`

Sets the text for the input box.

`setWrap(wrap)`

Sets whether the text should automatically wrap.

Functions

None.

DTObjectStore

The DTObjectStore package defines the Persistent and Root Persistent Object classes

For more information, see Persistence.

Classes

PersistentObject

Defines an object with properties that may be stored on the Desktop.com server. These properties may be numbers, strings, Booleans, or references to other POs.

All Persistent Objects are identified by a three-number tuple. The first number is the user ID, the second is the RPO ID, and the third is the child ID. Each user has their own user ID, and all of that person's POs start with their user ID as the first number in the tuple. The second number identifies an Object Group. Every PO within that group has the same first and second number in the tuple. The third number identifies the particular PO within the group, with the RPO always having a child ID of 0.

Inherits from DTObjectFramework.DTObject.

Constructor

`Constructor: PersistentObject(arg1, arg2)`

> `arg1` defines the user ID, and `arg2` the Root Persistent Object with which this PO will be associated.
>
> This constructor may be called in three different ways:
>
> `new PersistentObject(PORef, obj)` creates a PO using PORef and obj, which recreates the PO on the client side.
>
> `new PersistentObject(PO)` creates a NEW PO in the same RPO as PO.
>
> `new PersistentObject(PORef)` creates NEW PO using PORef.

It is important to note that this constructor should pass its two arguments to the PersistentObject constructor, which will in turn call `construct()` or `restore$()`, as appropriate.

Methods

childArray()
children()
deleteKey(k)
deleteSelf()
fetchChild$(key)
fetchChildren$()
fetchChildrenRecursive$()
isDeleted()
keyArray()
keyExists(k)
keys()
nonChildKeyArray()
nonChildKeys()
set(key, value)

`childArray()`

Returns an array with the names of all properties that refer to other `PersistentObject`s.

`children()`

Returns a hash with the names of all properties that refer to other `PersistentObject`s.

`deleteKey(k)`

Deletes the property with key `k`.

`deleteSelf()`

Deletes the persistent object itself.

`fetchChild$(key)`

Asynchronously fetches the object for field name `key` and then calls the callback.

Returns `{name, success, error}`.

`fetchChildren$()`

Asynchronously fetches all child objects of this Persistent Object from the server, then calls the callback (callback argument is `{success, error, errors}`). Because only some child objects may fail, the `errors` property of the callback's argument object is a hash with the child name as the key and the error as the value.

`fetchChildrenRecursive$()`

Asynchronously fetches all child objects, and their children recursively, then calls the callback. Because only some child objects may fail, the errors property of the callback's `arg` object is a hash with the child name as the key and the error as the value. Because fetching is recursive, error values in the errors hash may themselves be error objects of children. This indicates that the child itself came down fine, but one or more of its descendants did not.

Returns `{success, error, errors}`. If success is `false`, the `error` property is a number, and the `errors` property is an object.

`isDeleted()`

Returns `true` if the object is deleted; `false` if not.

`keyArray()`

Returns an array with all of the persistent property names.

`keyExists(k)`

Returns `true` if the property specified by its key (`k`) has been set; `false` if not.

`keys()`

Returns a hash with all of the persistent property names. This is meant to be used as:
`for(var k in po.keys()) { ... }`

`nonChildKeyArray()`

Return an array with the names of all properties that *do not* refer to `PersistentObject`s.

`nonChildKeys()`

Return a hash with the names of all properties that *do not* refer to `PersistentObject`s.

`set(key, value)`

Sets the property persistently.

RootPersistentObject

A RootPersistentObject (RPO) differs from a PersistentObject (PO) in that it acts as a container for other PersistentObjects, and may be placed in the filesystem.

All PersistentObjects except RPOs must be associated with a RootPersistentObject.

Inherits from DTObjectStore.PersistentObject.

Constructor

`RootPersistentObject(arg1,arg2)`

`arg1` defines the user ID, and `arg2` the ID for this RPO.

RootPersistentObjects may be created in the following ways:

`rpo = new RootPersistentObject()`; creates a new RootPersistentObject that does not live in the filesystem but can be placed there with `FileSystem.put()`.

`rpo = new RootPersistentObject(PORef)`; creates a new RootPersistentObject with a given ID.

Methods

childArray()
children()
construct()
keyArray()
keys()

`childArray()`

Returns an array of all public keys that are child objects.

`children()`

Returns a hash of all public keys that are child objects.

`construct()`
: Constructs an RPO.

`keyArray()`
: Returns an array of all public keys.

`keys()`
: Returns a hash of all public keys.

Functions

`getErrorString(err)`
: Returns the specified error as an English text string.

`makeSafe(s)`
: Returns a new string, which is a "safe" encoding of the string specified. "safe" means that it is a valid JavaScript identifier.

`set(obj, prop, value)`
: Sets a property/value pair for a persistent object. `obj` may be a reference to the PersistentObject, or a string representing the owner/id pair. `prop` is the property of the variable to set. `value` is a number, string, or PersistentObject that represents the value being set.

`unMakeSafe(s)`
: Returns the original string, which may be an invalid JavaScript identifier.

`update$(objs)`
: Updates the local copy of the object to the state of the server copy.

DTPane

The DTPane package defines the Pane class.

Classes

Pane

The Pane class is a subclass of Container that has the ability to manage its children (all Components) with a concept of ***focus***. At any one time, zero or one of a Pane's focusable children may have the Pane's focus. If there is a child in focus, it is only ***active*** if the Pane itself has focus within its parent Pane, and so on up to the Screen, which is the top-level Pane.

In tracing down the Pane/Component containment tree from its root at the Screen, the path along which Components are active is called the ***active path***. The deepest Component in the active path, the active component, is called the ***primary*** Component.

A Pane may have one of its child Components designated as its ***default focus***. A *strong* default focus (which may be `null`) attempts to take focus when a Pane becomes active. A *weak* default focus takes focus when a Pane becomes active and has no previously focused Component.

A ***focus ring*** is a stable ordering of focusable children in a Pane, used to define a serial path of user movement among Components.

For more information, see Focus.

Inherits from DTContainer.Container.

Constructor

`Pane(layout_manager)`

`layout_manager` (optional) specifies the layout manager to attach to the pane.

Actions

None.

Methods

addComponent(component, constraints, beforeComponent)
advanceFocus(direction)
handleKeyboardEvent(event)
removeComponent(component)
reorderComponent(component, beforeComponent)
setAllowBackgroundSteals(allow)
setDefaultFocus(component, strong)
setEnabled(enabled)
setFocus(component)
setFocusToEnd(whichEnd)

`addComponent(component, constraints, beforeComponent)`

Adds a `component` to the pane, placed before `beforeComponent` in the focus ring. `constraints` defines a constraints object to be passed to the layout manager's `addComponent` method. Note that not all layout managers require a constraints object.

`advanceFocus(direction)`

Advances focus through the focus ring in the direction specified: `1` for forward; `-1` for reverse.

`handleKeyboardEvent(event)`

Event handler for keyboard events. Pressing the tab key moves focus forward through the focus ring. Pressing shift-tab moves focus backward through the focus ring. All other keyboard events are ignored. This method is typically overridden to take application-specific action based on the defined keyboard events.

`removeComponent(component)`

Removes the specified `component` from the pane. Changes focus and removes the component from the focus ring, if necessary.

`reorderComponent(component, beforeComponent)`

Moves the specified `component` to the position before the `beforeComponent`.

`routeRefocusEvent(component, reason, event)`

Called by a child component when it has received a refocus event.

`setAllowBackgroundSteals(allow)`

Sets whether clicking on the Pane's background sets the focus to `null`. Default is `true`.

`setDefaultFocus(component, strong)`

Specifies the `component` to which focus will be set by default. `strong` specifies whether or not the default focus is strong. If `true`, an implicit refocus will change focus to the default; if `false`, it will change to the last focused component.

`setEnabled(enabled)`

Sets the enabled state of the component: `true` if enabled; `false` if not.

`setFocus(component)`

Changes focus to the specified component. Returns `true` if focus was changed successfully, `false` if not.

`setFocusToEnd(whichEnd)`

Sets focus to the end of the focus ring: `-1` if the beginning; `1` if the end of the ring.

Functions

None.

DTPersistentArray

The DTPersistentArray package defines the PersistentArray class.

Classes

PersistentArray

A PersistentArray object stores an array of information that is maintained on the server, and therefore persists from user session to user session. This may include information such as the user ID, and whether the application was launched and its window location the last time the Desktop.com session was ended.

For more information, see Persistence.

Inherits from DTObjectStore.PersistentObject.

Constructor

`PersistentArray(arg1,arg2)`

`arg1` defines the user ID, and `arg2` the RootPersistentObject with which this array is associated.

Methods

getLength()	set(index, value)
keyArray()	setLength(len)
keys()	sort(compare)
pop()	splice(array, start, delete_count, values)
push(value)	top()

`getLength()`

Returns the length of the array.

`keyArray()`

Returns an array of all public keys.

`keys()`

Returns a hash of all public keys.

`pop(a)`

Removes and returns the last element of the array.

`push(value)`

Pushes the `value` on the end of the array.

`set(index, value)`
: Sets the element at location `index` to the specified `value`.

`setLength(len)`
: Sets the length of the array.

`sort(compare)`
: Sorts the array based on `compare`, which is an optional function. Numeric or alphabetic input is the default.

`splice(array, start, delete_count, values)`
: Inserts and/or deletes elements in an array. A single call to `splice()` may insert items, delete items, or do both simultaneously.

  `array` is a reference to the array to work with. `start` is an integer that specifies a zero-based index within `array` at which `splice()` will begin. `delete_count` is an integer that specifies how many elements of `array` to remove. If `delete_count` is zero, no deletion is performed. `values` (optional) is an array of elements to insert into `array`. If `values` is omitted or an empty array, no insertion is performed. Elements following the location of insertion or deletion are shifted so that no gaps appear in `array`, and the `length` property of `array` is updated to reflect its new size.

  Returns a reference to `array`.

  For an example, see DTArray.

`top()`
: Returns the last element of the array.

Functions

None.

DTProgressBar

The DTProgressBar package defines the ProgressBar class.

Classes

ProgressBar

Defines a standard, graphic progress bar.

Inherits from DTContainer.Container.

Constructor

`ProgressBar()`

Actions

None.

Methods

getProgress()	setSize(w, h)
setProgress(p)	setWidth(w)

`getProgress()`

Returns the progress amount as a number between -1 and 1. If -1, the progress bar is inactive, and displays an animated picture that looks like a scrolling bar.

`setProgress(p)`

Sets the progress amount where p is a number between -1 and 1. If -1, the progress bar is inactive, and displays an animated picture that looks like a scrolling bar.

`setSize(w, h)`

Sets the width of the bar. The height is always DTProgressBar.ProgressBar.HEIGHT, no matter what h is passed in.

`setWidth(w)`

Sets the width of the bar in pixels.

Functions

None.

DTPrompt

The DTPrompt package defines the `DoPrompt$()` function.

Classes

None.

Functions

`DoPrompt$(text, input, where)`

Displays a dialog prompt window, which displays the given `text`, and the default value `input` (optional) in a user-input field. `where` (optional) specifies a reference to an object (either an Application or a Window) relative to which the dialog is to be centered and made modal. If `where` is omitted, the dialog will be centered on the screen and made system-modal. As window centering and modality is the most common, pass a Window reference for `where` unless there is some reason not to.

Returns the user's input value, or `null` if cancelled.

DTQuestion

The DTQuestion package defines the `AskQuestion$()` function.

Classes

None.

Functions

`AskQuestion$(text, where)`

Displays a dialog window with the specified text, and Yes, No, and Cancel buttons. Dialog windows are resized to fit the text. Text is wrapped where appropriate, and "\n" inserted in a line of text will define a line break. `where` (optional) specifies a reference to an object (either an Application or a Window) relative to which the dialog is to be centered and made modal. If `where` is omitted, the dialog will be centered on the screen and made system-modal. As window centering and modality is the most common, pass a Window reference for `where` unless there is some reason not to.

Returns `yes`, `no`, or `cancel`.

DTQueue

The DTQueue package defines the Queue class.

Classes

Queue

The Queue class defines an ordered array of objects.

Inherits from DTObjectFramework.DTObject.

Constructor

`Queue(max_size)`

`max_size` sets the maximum length of the queue.

Methods

deQ()
enQ(data)
getSize()
peek(n)
remove(n)

`deQ()`

Removes and returns the object from the end of the queue.

`enQ(data)`

Adds the given data to the queue.

`getSize()`

Returns the number of elements in the queue.

`peek(n)`

Returns the object in the queue at position number `n`, with the first position zero. By default, c=0.

`remove(n)`

Removes the object in the queue at position `n`.

Functions

None.

DTRadioButtonGroup

The DTRadioButtonGroup Package defines the RadioButtonGroup class.

Classes

RadioButtonGroup

Encapsulates a group of radio-button controls, which function like check boxes, except that only one item in the group may be selected at a time. It is also possible to have none of the items in the group selected.

A ColumnLayoutManager is automatically attached to the group. To configure a RadioButtonGroup's layout, call its `getLayoutManager()` method to retrieve a reference to the ColumnLayoutManager, then call methods of the layout manager. Radio button groups should not be sized with `setSize()`.

Radio button groups are non-circular Panes, meaning that keyboard-based focus advances will travel through the group and out the other side rather than wrapping around. (See Focus in the GUI Structure chapter for details.) Radio button groups interpret the up- and down-arrow keys, and the tab and shift-tab keys, as focus-advance keystrokes. Pressing the spacebar will select the item in the group that is in focus.

Inherits from DTPane.Pane.

See also DTBorderedRadioButtonGroup.BorderedRadioButtonGroup, and DTColumnLayoutManager.ColumnLayoutManager.

Constructor

`RadioButtonGroup()`

Actions

`changed`: selection has changed to a different item in the group.

Methods

addItem(text, data)
getCheckedItem()
getCheckedItemData()
setAllowUserUncheck(allow)
setCheckedItem(index)
setCheckedItemByData(data)
uncheckAllItems()

`addItem(text, data)`

Adds an item with the specified `text` to the group. Items appear listed in the order in which they are added. The `data` argument is a string or number that serves to identify the item being added. This value may be retrieved using `getCheckedItemData()` when the item is selected. If `data` is omitted, the `text` argument will be used for the item's data.

`getCheckedItem()`
: Returns the (zero-based) index of the currently selected item, or `null` if no item is selected.

`getCheckedItemData()`
: Returns the data associated with the currently selected item, or `null` if no item is selected.

`setAllowUserUncheck(allow)`
: Sets whether users are able to deselect all items in the group. When allow is `true`, users may deselect by clicking the selected item. Default is `false`.

`setCheckedItem(index)`
: Selects the item with the specified (zero-based) `index`.

`setCheckedItemByData(data)`
: Selects the item with the specified `data`.

`uncheckAllItems()`
: Resets the group to the state in which no item is selected.

Functions

None.

DTRectangle

The DTRectangle package defines the Rectangle class.

Classes

Rectangle

Defines a rectangular border object used to outline groups of related components.

Note that a rectangle used to enclose other components must be placed BEHIND them. If it is not, the components will be visible, but they will not respond to mouse events.

Inherits from DTContainer.Container.

Constructor

`Rectangle(Color)`
: If color is set to white, the border of the rectangle is set to white; if not, it is black.

Actions

None.

Methods

`setSize(w, h)`
: Sets the outer size of the rectangle, including its border, in pixels.

`setThickness(thickness)`
: Sets the thickness of the rectangle border in pixels.

Functions

None.

DTRendezvous

The DTRendezvous package defines the Rendezvous class.

Classes

Rendezvous

Defines a meta-callback object that waits for a number of blocking function calls in parallel, then returns the results in an array.

The following example will wait until 5 seconds elapse, or both asynchronous `get` calls succeed.

```
var rv = new DTRendezvous.Rendezvous();
foo.get("left", rv.register(0));
foo.get("right", rv.register(1));
var results = rv.timedWait$(5000)
```

The results array returned via the `wait()` or `timedWait$()` method contains all results reported by that time.

You may also supply a notifier callback to the constructor. This notifier is called when the Rendezvous gets a result with the result (see below). If the notifier wishes, it may call `rv.abort()` with a reason and key to abort the wait immediately.

It is possible to register a number of calls with a Rendezvous, `wait$()`, register more, then `wait$()`again.

The result returned from `wait$()` and `timedWait$()` is an object with the following properties:

- `success`: whether all calls succeeded
- `error`: if success is `false`, returns DTRendezvous.ERRNO.ERR_TIMEOUT, DTRendezvous.ERRNO.ERR_ABORT, DTRendezvous.ERRNO.ERR_INUSE, and DTRendezvous.ERRNO.ERR_SOMEERRS.
- `abort_reason`: the reason code passed to `abort()` if error is ERR_ABORT.
- `abort_key`: the key as above.
- `results`: all the results so far. Note that new results may appear in this object later.

Inherits from DTObjectFramework.DTObject.

Constructor

`Rendezvous(notify_cb)`

`notify_cb` defines an optional callback which is called whenever a result is reported. This callback receives one argument, which is an object with two properties: `rv`, the rendezvous; and `value`, the result returned.

Methods

abort(reason, key)	timedWait$(timeout)
register(key)	wait$()

`abort(reason, key)`

Signals an error. Should only be called from a notifier callback.

`register(key)`

Returns a callback used to submit a result. `key` (optional) should be a number or string.

`timedWait$(timeout)`

Waits until the time specified (in ms) and then passes results array to callback. Timeout of -1 means wait indefinitely.

`wait$()`

Waits indefinitely (or until `notify_cb` returns non-null), then passes results array to callback.

Functions

None.

DTScrollBar

The DTScrollBar package defines the ScrollBar class.

Classes

ScrollBar

Defines a scrollbar object, consisting of a bar with a moveable slider, and arrow buttons.

The slider size/position is not (necessarily) an integer value of pixels, which allows more precise adjustment of its position. (It is only converted to an integer when passed to `setPosition()` or `setSize()` for the slider.) This allows the developer, for example, to guarantee a 15-pixel virtual increment regardless of the real size of the scroll bar.

Inherits from DTContainer.Container.

Constructor

`ScrollBar(orientation)`

> `orientation` (required) may be either ScrollBar.HORIZ or ScrollBar.VERT to construct a horizontal or vertical scrollbar, respectively.

Actions

None.

Methods

getOrientation()	setSize(w, h)
getSliderSize()	setSliderSize(s)
positionComponents()	setSliderSizeProportional(s)

`getOrientation()`

> Returns the horizontal or vertical orientation of the scroll bar.

`getSliderSize()`

> Returns the width and height of the slider, in pixels. Sliders are always square, and therefore take only one size value.

`positionComponents()`

> Updates the window components based on the current position and size of the slider.

`setSize(w, h)`

> Sets the width and height of the scroll bar, in pixels.

`setSliderSize(s)`

> Sets the width and height of the slider, in pixels. Sliders are always square, and therefore take only one size value.

`setSliderSizeProportional(s)`
: Sets the proportional location of the slider. Valid input is a number between 0 and 1.

Functions

`getSliderPosition()`
: Returns the position of the slider in pixels.

`getSliderPositionProportional()`
: Returns the position of the slider as a real number between 0 and 1.

`getSliderSizeProportional()`
: Returns the proportional size of the slider. Valid input is a number between 0 and 1.

`setSliderPosition(p)`
: Sets the position of the slider in pixels.

`setSliderPositionProportional(p)`
: Sets the position of the slider as a real number between 0 and 1.

DTScrollingTextBox

The DTScrollingTextBox package defines the ScrollingTextBox class.

Classes

ScrollingTextBox

Defines an automatically scrolling text box, which has a vertical scrollbar, and resizes and rewraps itself automatically, in which developer or user defined text may be displayed.

A Scrolling Text Box has two sizes: its size, and its virtual size. Size is the component's absolute size, including any areas which might be beyond the displayed area and invisible to the user. Virtual size is that visible on the screen at any time.

Inherits from DTScrollPane.ScrollPane.

Constructor

`ScrollingTextBox()`

Actions

None.

Methods

changeVirtualSize(dw, dh)
getText()
setSize(w, h)
setText(text)
setVirtualSize(w, h)
setVirtualWidth(w)
toggleInput(allow)

`changeVirtualSize(dw, dh)`
: Changes the outer width and height of the virtual text box by the given number of pixels.

`getText()`
: Returns the user input text from the box.

`setSize(w, h)`
: Sets the visible outer width and height of the box.

`setText(text)`
: Sets the text to display in the box.

`setVirtualSize(w, h)`
: Sets the width and height of the virtual text box.

`setVirtualWidth(w)`
: Sets the width of the virtual text box.

`toggleInput(allow)`
: Allows the developer to turn input on or off.

Functions

None.

DTScrollPane

The DTScrollPane package defines the ScrollPane class.

Classes

ScrollPane

Defines a scroll pane (a window pane and an attached scrollbar), in which the visible size of the pane is smaller than the size of its content. Attached scroll bars allow the user to scroll through the entire pane.

Inherits from DTPane.Pane.

Constructor

`ScrollPane(layout_manager)`

`layout_manager` sets the manager to attach to the pane.

Actions

None.

Methods

addComponent(c)	removeComponents(c)
changeVirtualSize(dw, dh)	setBackgroundColor(color)
getComponent(index)	setHorizontalScroll(enabled)
getLayoutManager()	setScrollPosition(x, y)
getNumberComponents()	setSize(w, h)
getScrollPosition()	setVerticalScroll(enabled)
getVirtualSize()	setVirtualSize(w, h)
removeComponent(c)	setVirtualWidth(w)

`addComponent(c)`

Adds the specified component to the pane.

`changeVirtualSize(dw, dh)`

In Netscape, changes the scrollable width and height of the pane by the given number of pixels. In Internet Explorer, does nothing, as size is set automatically.

`getComponent(index)`

Returns the component defined by `index`.

`getLayoutManager()`

Returns the layout manager associated with the scroll pane.

`getNumberComponents()`
: Returns the number of components added to the scroll pane.

`getScrollPosition()`
: In Netscape, Returns the coordinates of the origin of the slider in the scrolling region. In Internet Explorer, returns `null`.

`getVirtualSize()`
: In Netscape, Returns the scrollable width and height of the scroll pane (the virtual size of the pane). In IE, returns the size of the visible region.

`removeComponent(c)`
: Removes the specified component from the pane.

`removeComponents(c)`
: Removes all components from the pane.

`setBackgroundColor(color)`
: In Netscape, sets the color of the scrollbar, and the background color of the pane. In Internet Explorer, does nothing. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setHorizontalScroll(enabled)`
: In Internet Explorer, enables horizontal scrolling. `true` by default.

`setScrollPosition(x, y)`
: In Netscape, sets the x and y coordinates of the origin of the slider in the scrolling region. In Internet Explorer, has no effect

`setSize(w, h)`
: In Netscape, sets the viewable width and height of the scroll pane. In Internet Explorer, does nothing, as size is set automatically.

`setVerticalScroll(enabled)`
: In Internet Explorer, enables vertical scrolling. `true` by default.

`setVirtualSize(w, h)`
: In Netscape, sets the scrollable width and height of the scroll pane. In Internet Explorer, does nothing, as the virtual size is set automatically.

`setVirtualWidth(w)`
: In Netscape, sets the scrollable width of the pane. In Internet Explorer, sets the viewable width of the pane.

Functions

None.

DTSlideDialog

The DTSlideDialog package defines the SlideDialog class.

Classes

SlideDialog

Displays a dialog window containing a series of panes, only one of which is visible at a time. The SlideDialog includes Next and Back buttons, which allow the user to navigate through the series.

Inherits from DTTransientDialogWindow.TransientDialogWindow.

Constructor

`SlideDialog()`

Actions

None.

Methods

addPane(pane, title)
backPane()
getPaneSize()
getVisibleIndex()
getVisiblePane()
nextPane()
setVisiblePane(idx)

`addPane(pane, title)`
: Adds a pane with the given title.

`backPane()`
: Displays the previous pane.

`getPaneSize()`
: Returns the width and height of the pane.

`getVisibleIndex()`
: Returns the index of the visible pane.

`getVisiblePane()`
: Returns a key to the visible pane.

`nextPane()`
: Displays the next pane in the series.

`setVisiblePane(idx)`
: Sets the visible pane.

Functions

None.

DTSlider

The DTSlider package defines the Slider class.

Classes

Slider

Sliders are user-input controls that accommodate numeric values between two endpoints. Graphically, a slider is a track with a button that may be dragged left or down (for decreasing values) and right or up (for increasing values) along the track. Clicking along the track away from the slider button will "bump" the slider along the track in that direction.

Sliders are not required to have integer values or limits. They may be forced to by specifying integral limits and an integral increment size, using an incremental slider (see DTIncrSlider).

Sliders do not support dynamic dragging; they generate value change actions only at the end of interactive drags, not during them. Continuous sliders round to the nearest pixel for both dragging and bumping increments.

A slider cannot accept input of more than its maximum or less than its minimum.

The widget's breadth (size along the axis perpendicular to the track) cannot be changed; the relevant argument to `setSize()` is ignored.

Note that in vertical sliders, a higher y value indicates a lower numerical value for the slider.

Inherits from DTContainer.Container.

Constructor

`Slider()`

Actions

drag finish: "changed"
bump click: "changed"
setValue(): "changed"

Methods

getLimits()
getOrientation()
getValue()
setLimits(minVal, maxVal)
setOrientation(orientation)
setSize(w, h)
setValue(value)

`getLimits()`

Returns the minimum and maximum values for the slider.

`getOrientation()`
: Returns the orientation of the slider.

`getValue()`
: Returns the value for the slider position. This method may return an undesirable number, so callers should round off if a cleaner number is needed. If the desired rounding granularity causes distinct pixel positions to map to the same value, use an incremental slider. Note that using an incremental slider with increments that map to less than a few pixels may work poorly.

`setLimits(minVal, maxVal)`
: Sets the minimum and maximum values for the slider.

`setOrientation(orientation)`
: Sets the vertical or horizontal orientation of the slider.

`setSize(w, h)`
: Sets the width and height of the slider.

`setValue(value)`
: Sets the value for the slider position.

Functions

None.

DTStack

The DTStack package defines the Stack class.

Classes

Stack

Defines a simple stack object.

Constructor

`Stack()`

Actions

None.

Methods

getSize()	push(element)
peek(c)	remove(n)
pop()	

All Stack methods return `undefined` if the parameters are out of range.

`getSize()`
: Returns the number of elements in the stack.

`peek(c)`
: Returns the object on the stack at position number `c`, with the first position zero. By default, `c=0`.

`pop()`
: Pops and returns the top element.

`push(element)`
: Pushes the element `e` on the end of the array.

`remove(n)`
: Removes the top `n` things from the stack, or all things if no `n` is passed in.

Functions

None.

DTStyle

The DTStyle package defines the Style class.

Classes

Style

Defines a style object. `Component.setStyles()` may be used to set the style.

These methods should be used only in developer implemented, low-level components.

Inherits from DTObjectFramework.DTObject.

Constructor

`Style()`

Methods

`get(prop)`
: Returns the style sheet properties for a component.

`set(prop, value)`
: Sets the style sheet properties for a component.

Functions

None.

DTTableLayoutConstraints

The DTTableLayoutConstraints package defines the TableLayoutConstraints class.

Classes

TableLayoutConstraints

Defines the parameters used to construct a table layout.

Constructor

`TableLayoutConstraints()`

Actions

None.

Methods

`setDefaults()`
: Sets the default layout constraints for the layout manager.

Functions

None.

DTTableLayoutManager

The DTTableLayoutManager package defines the TableLayoutManager class.

Classes

TableLayoutManager

Defines a TableLayoutManager, which may be associated with any pane or window. Desktop.com tables closely resemble HTML tables, in that individual rows or columns may have differing heights or widths, respectively. Padding for cells may be set by individual cell.

TableLayoutManagers differ from GridLayoutManagers both in the flexibility of cell and row sizing, and in that a table will distribute available extra space to cells depending on their weight properties.

See also GridLayoutManager.

Inherits from DTLayoutManager.LayoutManager.

Constructor

`TableLayoutManager()`

Methods

addComponent(component, constraints)	layoutComponents()
getColumns()	setColumns(ncols)
getRows()	setRows(nrows)

`addComponent(component, constraints)`

Adds a component to the table using the specified `constraints`, which may take one or more of the following values:

`fill`: specifies if the component should be stretched vertically, horizontally, or both to fill the cell. Valid input includes FILL_NONE, FILL_BOTH, FILL_HORIZONTAL, and FILL_VERTICAL.

`weight_x`: the x sizing weight.

`weight_y`: the y sizing weight.

`span_x`: the number of columns the component may cross horizontally.

`span_y`: the number of columns the component may cross vertically.

`pad_x`: the horizontal padding.

`pad_y`: the vertical padding.

`loc_x`: the column position of the component.

`loc_y`: the row position of the component.

`getColumns()`

Returns the number of columns in the table.

`getRows()`
: Returns the number of rows in the table.

`layoutComponents()`
: Lays out the components for the table.

`setColumns(ncols)`
: Sets the number of columns in the table.

`setRows(nrows)`
: Sets the number of rows in the table.

Functions

None.

DTTabView

The DTTabView package defines the TabView class.

Classes

TabView

Defines a tabbed window, with multiple pane components, only one of which is visible at any given time. This class defines a widget which allows users to switch between window panes by clicking on tabs.

Note: Do not call `setSize()` on panes added to TabView, as TabView will automatically size any added panes. To get the size TabView uses (for example, to adjust the size of a component contained within a pane added to the TabView), call `getSize()` on an individual pane.

Inherits from DTPane.Pane.

Constructor

`TabView()`

Actions

`tabbed`: a new tab has been clicked.

Methods

getActivePane()	removePane(num)
getLabel(num)	setActivePane(activePane)
getNumPanes()	setLabel(num, label)
getPane(num)	setPane(num, pane)

`addPane(pane, label)`
: Adds the specified `pane` to the tab view, and labels the added tab with the `label` provided.

`getActivePane()`
: Returns the active pane in the tab view.

`getLabel(num)`
: Returns the tab label for the pane specified by `num`.

`getNumPanes()`
: Returns the number of panes added to the tab view.

`getPane(num)`
: Returns the pane specified by `num`.

`removePane(num)`
: Removes the pane specified by `num`.

`setActivePane(activePane)`
: Sets the specified pane to be active.

`setLabel(num, label)`
: Sets the tab label of the pane specified by `num`.

`setPane(num, pane)`
: Replaces the pane specified by `num` with the new pane (`pane`).

Functions

None.

DTTextBox

The DTTextBox package defines the TextBox class.

Classes

TextBox

Defines a box in which text is displayed.

Inherits from DTComponent.Component.

Constructor

`TextBox(text, alignment)`
: Specifies the `text` and its `alignment` in the box.

Actions

None.

Methods

getAlignment()
getText()
setAlignment(alignment)
setColor(color)
setText(text)

`getAlignment()`
: Returns the alignment for the text within the box.

`getText()`
: Returns the text from the box.

`setAlignment(alignment)`
: Sets the alignment for the text within the box. Valid input includes TextBox.LEFT, TextBox.RIGHT and TextBox.CENTER.

`setColor(color)`
: Sets the color of the text. `color` is defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

`setText(text)`
: Sets the text string to be used for the box.

Functions

None.

DTTextImageLabel

The DTTextImageLabel package defines the TextImageLabel class.

Classes

TextImageLabel

Defines a text label object, with an optional image included above or to the left of the text. An actionListener for events such as "click" and "double click" may be associated with this label.

Inherits from DTTextBox.TextBox.

Constructor

`TextImageLabel(text, image, alignment, tooltip)`

Specifies the `text` and `image` to be included for the label, with the given `alignment` and `tooltip` to display for mouse-over.

Actions

None.

Methods

getImage()
setImage(image)
setText(text, num_breaks)

`getImage()`

Returns the path to the image.

`setImage(image)`

Sets the image to be displayed with the text.

`setText(text, num_breaks)`

Sets the text for the component. `num_break` specifies the number of new lines that will be inserted between the image and the text.

Functions

None.

DTTextInputBox

The DTTextInputBox package defines the TextInputBox class.

Classes

TextInputBox

Defines a text input box, which may be used to gather user input. The box displays multi-line text which the user may edit. Word wrapping is done automatically.

Inherits from DTComponent.Component.

Constructor

`TextInputBox()`

Actions

None.

Methods

deleteCharLeft()
getText()
insertChar(c)
setText(t)
wrap_lines(l, check_for_underflow, force_left_to_right)

`deleteCharLeft()`
: Deletes the character to the left of the cursor.

`getText()`
: Returns the string of text currently in the box.

`insertChar(c)`
: Inserts the character `c` at the current cursor location.

`setText(t)`
: Sets the text in the box to the string `t`, and word wraps if necessary.

`wrap_lines(l, check_for_underflow, force_left_to_right)`
: Wraps the lines of text. `l` is the line from which to begin the wrap; `check_for_underflow` checks to see if the previous line is full; and `force_left_to_right` forces the wrapping algorithm to work from left to right, for greater efficiency.

Functions

`handleMouseEvent(e)`
: Called when a mouse event is received.

DTTextInputBox2

The DTTextInputBox2 package defines the DTTextInputBox2 class.

Classes

TextInputBox2

Defines a text input box, which may be used to gather user input. The box displays multi-line text which the user may edit. Carriage returns are recorded. Word wrapping is done automatically.

Inherits from DTComponent.Component.

Constructor

`TextInputBox2()`

Actions

`text_changed`: the text in the Box has been changed.

Methods

deleteCharLeft()	insertChar(c)
getText()	setText(t)

`deleteCharLeft()`
: Deletes the character to the left of the cursor.

`getText()`
: Returns the string of text currently in the box.

`insertChar(c)`
: Inserts the character `c` at the current cursor location.

`setText(t)`
: Sets the text in the box to the string `t`, and word wraps if necessary.

Functions

None.

DTTextInputField

The DTTextInputField package defines the TextInputField class.

Classes

TextInputField

Defines a one-line text input field (which may be placed in a pane or other components), implemented using the browser's native HTML <input type=text> component.

Inherits from DTNativeComponent.NativeComponent.

Constructor

`TextInputField(t)`

`t` (optional) defines the initial, editable text.

Actions

`changed:` focus is lost and text has changed.

Methods

getText()
setEnterKeyMode(on)
setFocusable(focusable)
setLength(length)
setSize(w, h)
setText(t)
setWidth(w)

`getText()`

Returns the text from the field.

`setEnterKeyMode(on)`

Sets whether the enter key will cause an event to occur.

`setFocusable(focusable)`

Sets whether the component may receive focus.

`setLength(length)`

Sets the width of the field in number of characters.

`setSize(w, h)`

Sets the width and height of the field in pixels.

`setText(t)`

Sets the text for the field, which may be user-defined.

`setWidth(w)`
: Sets the width of the field in pixels.

Functions

None.

DTTextLabel

The DTTextLabel package defines the TextLabel class.

Classes

TextLabel

Defines a single-line version of the TextBox object. It displays a single line of text, which will not wrap or resize the component in which it is placed.

Inherits from DTTextBox.TextBox.

Constructor

`TextLabel(text, alignment)`

Specifies the `text` for the label, and the text's `alignment` within the label.

Actions

None.

Methods

`setSize(w, h)`

Sets the size of the text label. Text will be truncated if it is beyond the size specified here.

Functions

None.

DTTextTreeView

The DTTextTreeView package defines the TextTreeView class.

Classes

TextTreeView

Defines an object which allows users to view a hierarchy in the form of a tree. The hierarchy is stored by adding paths (using `addPath()`) where /foo/bar/bag represents bag as a child of bar as a child of foo. Two paths, /foo/bar and /foo/bag represents bag and bar as equal children of foo.

Note that this class may be used to represent any sort of hierarchy: filesystems are only one possibility. In the first example above, /foo/bar/bag may represent both foo as father to bar, who is father to bag, making foo bag's grandfather; as well as bag as a folder within bar, which is in turn a folder within foo.

Inherits from DTScrollPane.ScrollPane.

Constructor

`TextTreeView()`

Actions

None.

Methods

addPath(path, icon1, data, icon2, hide, delimiter, node_icon, no_update)
eraseBranch(node)
getNodeChild(node, child)
getNodeCollapsed(node)
getNodeData(node)
getNodeFromPath(path, delimiter)
getNodeIcon(node)
getNodeText(node)
getPathFromNode(node)
getSelectedData()
getSelectedNode()
hideChildNodes(node)
removePath(path, delimiter)
setNodeCollapsed(node, collapsed)
setNodeData(node, data)
setNodeIcon(node, icon)
setNodeText(node, text)
setSelectedData(data)
setSelectedNode(node)
setTopNode(text, icon)
update()
updateTreeChildren()
updateTreeSelectionIndicator()

`addPath(path, icon1, data, icon2, hide, delimiter, node_icon, no_update)`

Adds a path to the tree, using the following arguments:

`path`: creates a path with all missing nodes from the root to the leaf. `path` is delimited by the string specified in `delimiter` (default is "`/`" if unspecified).

`icon1`: defines the default icon image to display with the new node (applies only to the leaf node being created)

`icon2`: (optional) specifies the image to use when the node is selected.

`data`: user specified data stored with the new node data is any object which a user might want to attach to the node corresponding to a path. This may be retrieved for later use by `getNodeData`.

`hide`: hides the node if `true`.

`no_update`: if set to `true`, the view is updated after the call. If the property exists, the text displayed with the node is `data.name`; if `data.name` does not exist, it is the name of the node as specified in the path.

`eraseBranch(node)`

Removes a branch of the tree and all its associated components, beginning with the given node. The node itself is not removed from the view. If one of the child nodes is selected, its parent node is also selected.

`getNodeChild(node, child)`

Returns the child of a node with the name specified by a string. Returns `null` if a child with that name does not exist.

`getNodeCollapsed(node)`

Determines if a node is collapsed.

`getNodeData(node)`

Returns the data associated with the specified `node`.

`getNodeFromPath(path, delimiter)`

Returns the node in the view corresponding to the specified path. `delimiter` is the separation in a path. Default is "`/`" (which specifies an absolute path).

`getNodeIcon(node)`

Returns the icon associated with the node.

`getNodeText(node)`

Returns the text associated with the node.

`getPathFromNode(node)`

Returns the path string in the form rootnode/path/to/node.

`getSelectedData()`

Returns the user specified data associated with the active node.

`getSelectedNode()`

Returns the selected node.

`hideChildNodes(node)`
: Recursively hides all children of the specified node, as well as the node itself.

`removePath(path, delimiter)`
: Removes a path from the tree. If the node specified by `path` exists, it will be removed (along with all of its children).

`setNodeCollapsed(node, collapsed)`
: Collapses or explodes the specified node.

`setNodeData(node, data)`
: Sets the data associated with a node. This method is provided for users' convenience, and allows them to attach data to nodes in the hierarchy.

`setNodeIcon(node, icon)`
: Sets the icon associated with the node.

`setNodeText(node, text)`
: Sets the text associated with the node.

`setSelectedData(data)`
: Sets the data associated with the active node.

`setSelectedNode(node)`
: Sets the selection to the specified node.

`setTopNode(text, icon)`
: Sets the text and icon for the top-level node.

`update()`
: Updates the view of the text fields' sizes and locations for the tree.

`updateTreeChildren()`
: Updates the view of the text fields' sizes and locations for the tree and its children.

`updateTreeSelectionIndicator()`
: Sets the selection indicator component to the correct size and location.

Functions

None.

DTTimer

The DTTimer package defines the Timer class.

Classes

Timer

Defines a timer object to be associated with a component, which invokes a callback after a defined amount of time.

Inherits from DTObjectFramework.DTObject.

Constructor

`Timer(callback, interval, repeat)`

> `interval` sets the number of seconds after which the callback should be called.
> `repeat`, if `true`, repeats the callback every defined interval of seconds, for the length of time the processor runs. (To prevent the callback from repeating, create a non-repeating timer whose callback restarts the timer.)

Methods

isRunning()	start()
setInterval(milli)	stop()

`isRunning()`

> Returns whether the timer is running.

`setInterval(milli)`

> Sets the timer interval in milliseconds. Does not affect running timers.

`start()`

> Starts the timer. By default, timers are created not running.

`stop()`

> Stops the timer without calling the callback. Note that it is possible for the callback to have already been called.

Functions

`sleep$(milli)`

> Sets the timer to sleep the given number of milliseconds.

DTTransientDialogWindow

The DTTransientDialogWindow package defines the DTTransientDialogWindow class.

Classes

TransientDialogWindow

Defines a class of Windows that can be created without an associated Application. Defines a temporary dialog, which may be used for any mutable application function. Dialog windows are created empty: any desired components may be added.The window's visual state will not be saved across Desktop sessions.

Inherits from DTDialogWindow.DialogWindow.

Constructor

`TransientDialogWindow()`

Actions

None.

Methods

autoLocation(arg)	enable$(arg)
close(arg)	wait$()

`autoLocation(arg)`

Automatically locates the window according to its size and the given Application, Window, or, by default, the screen. `arg` (optional) specifies a reference to an object (either an Application or a Window) relative to which the dialog is to be centered and made modal. If `arg` is omitted, the dialog will be centered on the screen and made system-modal. As window centering and modality is the most common, pass a Window reference for `arg` unless there is some reason not to.

`close(arg)`

Closes the window. If a callback is defined with `wait$()`, it is called with the given argument.

`enable$()`

Displays the window on the screen by calling the window's draw method.

`wait$()`

Waits until the window is closed, then returns the argument passed by `close(arg)`.

Functions

None.

DTUserData

The DTUserData package provides an object used to encapsulate user information such as their name and email address.

Classes

None.

Functions

`getLogin()`
: Returns the current user's login. (This function is similar to the "whoami" function in UNIX.)

`getUID()`
: Returns the current user's UID (user identification number).

`getUserInfo$()`
: Returns the user information as an object with four data members: `success` (`true` or `false`); `first`, the user's first name as a string; `last`, the user's last name as a string; and `email`; the user's email address.

`loginToUID$(login)`
: Converts a login name to UID number, where UID is the hash key of the user on the server (their reference number).

DTWindow

The DTWindow package defines the Window class.

Classes

Window

The base inner window used as the prototype for all application content panes in a managed window. Should not be instantiated directly, but its methods are inherited by other object classes.

Inherits from DTPane.Pane.

Constructor

`Window()`

Actions

None. Actions are delivered from the WindowFrame associated with the window, and include `closed`, `resized`, `moved`, and `iconized`.

Methods

close()
createVDO(parent_VDO)
destroyVDO(parent_VDO)
doneLoading()
getApp(app)
getLocation()
getTag()
init()
loading()
recreate(VDO)
restore()
setDrawMethod(method)
setLocation(x, y)
setRestoreMethod(method)
setSkipTaskBar(val)
setTag(tag)
setVisible(visible)
setWindowFrame(wRef)

`close()`
: Closes the window.

`createVDO(parent_VDO)`
: Creates a new visual display object with the parent object `parent_VDO`.

`destroyVDO(parent_VDO)`
: Destroys the visual display object with the parent object `parent_VDO`.

`doneLoading()`
: Called when the window is finished loading, just prior to the application calling itself.

`getApp()`
: Returns the application associated with the window.

`getLocation()`
: Returns the screen location of the window as an object with the properties `x` and `y`.

`getTag()`
: Returns a string containing the tag assigned to the window's visual display object (VDO), which contains information about the visual state of the window. The tag is set when the application's `addWindow()` method is called. `getTag()` is typically called when the user closes a window (or performs some other window action) to determine the action to be taken based on which window has been closed or manipulated.

`init()`
: Initializes the window.

`loading()`
: Displays an in-pane method between the painting of the window and the painting of the initial application components.

`recreate(VDO)`
: Called to restore a window's visual state on redraw.

`restore()`
: An optional method, which restores the paint style of the window. `draw()` is used if `restore()` is not present.

`setDrawMethod(method)`
: Sets the draw method used to paint the inner contents of the window.

`setLocation(x, y)`
: Specifies the `x` and `y` coordinates for the location of the window.

`setRestoreMethod(method)`
: Sets the draw method used to restore the window, which may differ from that used at startup. This method is optional. If not supplied, the method set by `setDrawMethod()` is used.

`setSkipTaskBar(val)`
: Specifies whether or not the window is visible in the task bar. `true` by default.

`setTag(tag)`
: Sets the unique `tag` string for the window, used to differentiate between windows owned by an application.

`setVisible(visible)`
: Sets whether the window is visible: `true` if visible (default), `false` if not.

`setWindowFrame(wRef)`
: Sets the WindowFrame object to be associated with this window, by reference.

Functions

None.

DTWrappingTextBox

The DTWrappingTextBox package defines the WrappingTextBox class.

Classes

WrappingTextBox

Defines a window with developer-defined or user-input text, which will automatically wrap when necessary.

Inherits from DTTextInputBox2.TextInputBox2.

Constructor

`WrappingTextBox()`

Actions

None.

Methods

handleKeyboardEvent(e)
rewrap_lines()
setSize(w, h)
setWidth(w)
toggleInput(allow)

`handleKeyboardEvent(event)`
: Called when a keyboard event is received.

`rewrap_lines()`
: Rewraps the text lines.

`setSize(w, h)`
: Sets the width and height for the box.

`setWidth(w)`
: Sets the width for the box.

`toggleInput(allow)`
: Allows the developer to turn input on or off.

Functions

None.

Appendix I: The Console commands

This appendix is provided as a quick reference guide to commands available in the Desktop.com Console, our command line interface application. All definitions are also available from the console itself. Simply type "help" and press enter for a list of available commands, or type "help command_name" for information on specific commands.

For all examples:

- The Console is case-sensitive.
- Italics denote user defined parameters.
- Items in brackets are optional.

`basename`: Prints the base name of path.
example: "`basename` *`path`*"

`browse`: Opens a web site.
example: "`browse`"

`canonical`: Prints the canonicalized name of path.
example: "`canonical` *`path`*"

`cat`: Displays file contents.
example: "`cat` *`file1`* [*`file2 ... fileN`*]"

`cd`: Changes the working directory.
example: "`cd [`*`dir`*`]`"

`chmod`: Changes permissions for a directory or sharelink.
example: "`chmod (-n|who (what|-))` *`sharepath`*"

`chpass`: Changes the current users password.
example: "`chpass` *`old_password new_password`*"

`clear`: Clears the screen.
example: "`clear`"

`compile`: Compiles a JSP file into a JSO file.
example: "`compile` *`JSP_filename JSO_filename`*"

`dirname`: Prints the dir name of path.
example: "`dirname` *`path`*"

`download`: Downloads a file.
example: "`download`"

`eval`: Evaluates a JavaScript expression.
example: "`eval` *`expression`*"

`exec`: Executes a compiled script.
example: "`exec` *`object_file`*"

`getlinks`: Displays links for a path.
example: "`getlinks` *`path`*"

`getuid`: Prints the UID of the requested user.
example: "`getuid` *`username`*"

`getuserinfo`: Prints the current users first name, last name, and email address.
example: "`getuserinfo`"

`help`: Lists available commands or describes the given command.
example: "`help [`*`command`*`]`"

`history`: Displays the command history.
example: "`history [`*`n`*`]`"

`id`: Prints the current user id.
example: "`id`"

`ln`: Creates a file link.
example: "`ln (-s|-m)` *`to_file from_file`*"
(with "–s" for a symbolic link, and "–m" for a smart link)

`load_package`: Imports a package into the code frame.
example: "`load_package`"

`log`: Writes a log message to the server.
example: "`log` *`message`*"

`ls`: Displays the contents of the current directory or specified directories.
example: "`ls [-l|-p|-t] [`*`dir1`* `[`*`dir2 ... dirN`*`]]`"

`mkdir`: Creates a directory (ies).
example: "`mkdir [-s]` *`dir1`* `[`*`dir2 ... dirN`*`]`"

(Including "–s" creates a symbolic path as well.)

`mv`: Renames a file.

example: "`mv from_filename to_filename`"

`nuke`: Unloads package(s) from the package manager.

example: "`nuke package_name1 [package_name2 ... package_nameN]`"

`open`: Opens a file with an application associated with the given type.

example: "`open filename [appname|null] [appargs]`"

`ps`: Lists running applications.

example: "`ps`"

`pwd`: Displays the current working directory.

example: "`pwd`"

`rm`: Removes files.

example: "`rm file1 [file2 ... fileN]`"

`rmdir`: Removes directories.

example: "`rmdir [-s] dir1 [dir2 ... dirN]`"

(Including "–s" removes symbolic paths as well.)

`run`: Runs an application.

example: "`run appname [appargs]`"

`seticon`: Sets or clears the icon for a path.

example: "`seticon path (type iconPath) | X`"

`share`: Creates a share link.

example: "`share path sharepath`"

`sharestatus`: Shows sharing info for a path.

example: "`sharestatus path`"

`tmpname`: Prints a filename that does not exist in path.

example: "`tmpname [prefix [path]]`"

`touch`: Creates an empty file.

example: "`touch file1 [file2 ... fileN]`"

`unshare`: Removes a sharelink or all sharelinks to a target.
example: "`unshare (path|sharepath)`"

`unwatch`: Unregisters a directory watcher.
example: "`unwatch watchnum`"

`upload`: Uploads a file.
example: "`upload`"

`uptime`: Shows uptime on the A server.
example: "`uptime`"

`watch`: Registers for updates regarding a directory.
example: "`watch path`"

`whoami`: Prints the username of the current user.
example: "`whoami`"

Appendix II: KeyEvent Constants

This appendix lists all defined constants for the KeyEvent class. The keyboard input to which these constants refer is self-explanatory, and is loosely arranged here as the standard keyboard is arranged: top to bottom, and left to right.

All number keys, 0-9, in the format: `KeyEvent.kVK_0`

plus

```
KeyEvent.kVK_BACK_QUOTE
KeyEvent.kVK_MINUS
KeyEvent.kVK_EQUALS
```

All Shift+number key combinations:

```
KeyEvent.kVK_TILDE
KeyEvent.kVK_EXCLAMATION
KeyEvent.kVK_AT
KeyEvent.kVK_POUND
KeyEvent.kVK_DOLLAR
KeyEvent.kVK_PERCENT
KeyEvent.kVK_CARET
KeyEvent.kVK_AMPERSAND
KeyEvent.kVK_STAR
KeyEvent.kVK_OPEN_PAREN
KeyEvent.kVK_CLOSE_PAREN
KeyEvent.kVK_UNDERLINE
KeyEvent.kVK_PLUS
```

All letter keys, a-z, in the format: `KeyEvent.kVK_A_LOWER`.

All Shift+letter key combinations, A-Z, in the format: `KeyEvent.kVK_A`.

All other typographic input:

```
KeyEvent.kVK_OPEN_CURLY
KeyEvent.kVK_CLOSE_CURLY
KeyEvent.kVK_PIPE

KeyEvent.kVK_OPEN_BRACKET
KeyEvent.kVK_CLOSE_BRACKET
KeyEvent.kVK_BACK_SLASH

KeyEvent.kVK_COLON
KeyEvent.kVK_SEMI_COLON
KeyEvent.kVK_DOUBLE_QUOTE
KeyEvent.kVK_SINGLE_QUOTE

KeyEvent.kVK_LESS_THAN
KeyEvent.kVK_GREATER_THAN
KeyEvent.kVK_QUESTION

KeyEvent.kVK_COMMA
KeyEvent.kVK_PERIOD
KeyEvent.kVK_FORWARD_SLASH

KeyEvent.kVK_SPACE
KeyEvent.kVK_BACKSPACE
KeyEvent.kVK_TAB
KeyEvent.kVK_ENTER
```

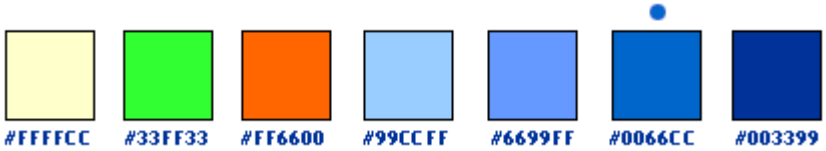
And special computer input keys:

```
KeyEvent.kVK_ESCAPE

KeyEvent.kVK_UP_ARROW
KeyEvent.kVK_DOWN_ARROW
KeyEvent.kVK_LEFT_ARROW
KeyEvent.kVK_RIGHT_ARROW

KeyEvent.kVK_HOME
KeyEvent.kVK_END
KeyEvent.kVK_PAGE_UP
KeyEvent.kVK_PAGE_DOWN
KeyEvent.kVK_DELETE
KeyEvent.kVK_INSERT
```

Appendix III: Color Values

Note that all color values within the Desktop.com development environment are defined as a 6-digit hex string which begins with a "#" character, such as "#00CC99."

The following Desktop.com color palette values are provided for your convenience. #0066CC is marked as the dominant color within the Desktop.com brand.



Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W