

# Intermediate Swing

Skill Level: Introductory

[Michael Abernethy \(mabernet@us.ibm.com\)](mailto:mabernet@us.ibm.com)

Author

29 Jun 2005

This tutorial builds on Introduction to Swing, which introduced the basics of Swing programming and the flight reservation system application. In this hands-on tutorial, Swing programmer Michael Abernethy walks you through more advanced Swing techniques like writing thread-safe code, building custom components, and customizing the look and feel to create a more polished and powerful UI.

## Section 1. Before you start

### About this tutorial

This tutorial is for those of you who have experience in putting together Swing applications, but would like to build on that knowledge with some more advanced techniques -- things that you might not be able to grasp just by looking at the Swing API. If you are thinking of taking this tutorial, you should be familiar with basic Swing concepts, such as Swing UI widgets, layouts, events, and data models. If you still think you need to review those concepts, be sure to review the [Introduction to Swing](#) tutorial, which covers all these areas and gives you the background you need to begin this tutorial.

During the course of this tutorial, you will be introduced to aspects of Swing beyond the basic components and applications. These areas of study, while more difficult to learn and grasp, are also more powerful and allow you to create better applications. The more advanced Swing concepts covered in this tutorial are:

- Understanding the JTable, and some of its many confusing and difficult concepts
- Writing thread-safe Swing code
- Creating a custom component

- Creating a completely custom look and feel

## Tools and source downloads

To complete this tutorial, you'll need the following:

- [JDK 5.0](#).
  - An IDE or text editor. I recommend [Eclipse](#) (see [Resources](#) for more information on Eclipse).
  - The [swing2.jar](#) for the flight reservation system.
- 

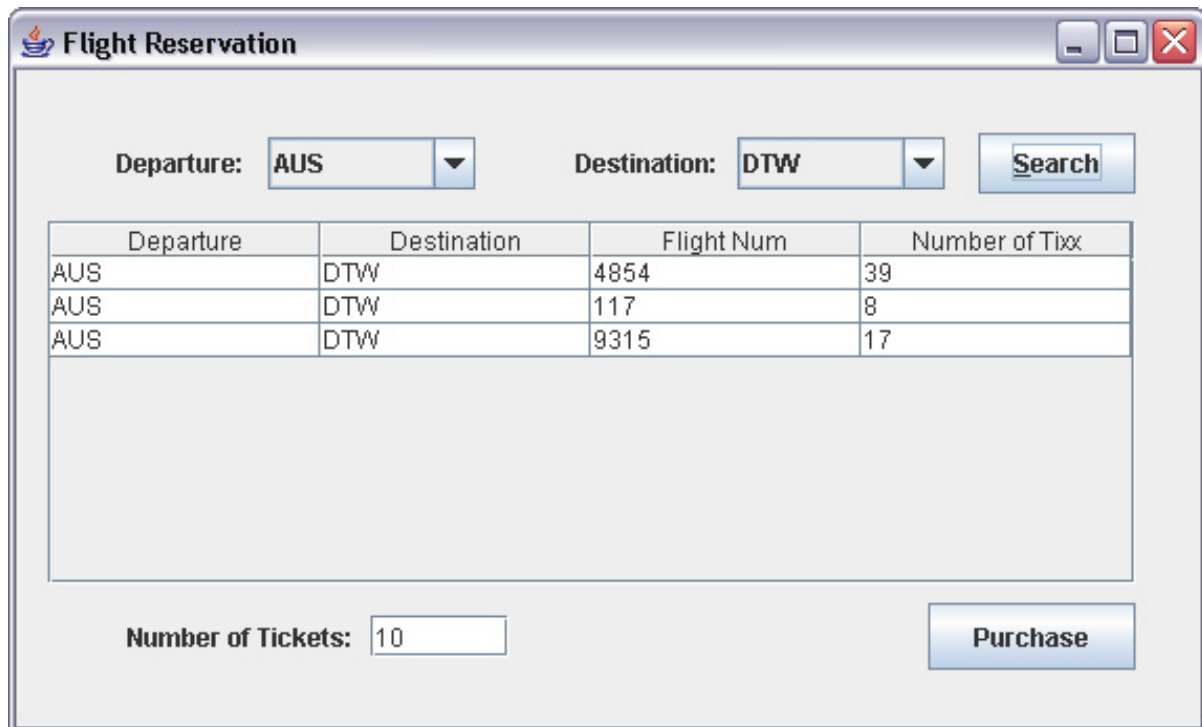
## Section 2. The sample application

### Refresher

Before you move on to the new concepts in this tutorial, let's revisit the sample application you started in the previous tutorial, the flight reservation system. For those of you that skipped the [first tutorial](#), this summary should help you quickly grasp what the application is about.

If you'll recall, the application tries to model a flight reservation system that would sit on your desktop. It allows the user to select a departure and destination city, and then search for flights that match the search criteria. These flights are displayed in a table, and the user can select the desired flight and purchase any number of tickets.

The application is very basic at this point; just by looking at it, you should be able to understand its role, as well the current limitations that we will address in this tutorial.



Departure	Destination	Flight Num	Number of Tixx
AUS	DTW	4854	39
AUS	DTW	117	8
AUS	DTW	9315	17

## Limitations of the application

Looking at the application as it exists now, you may think that it has everything it needs to work. Well, that's one way of looking at it -- it does have everything it needs to work, but it doesn't have everything it needs to work well. This is what separates the bad applications from the good ones -- the finer points that make the UI experience well-rounded. Although the application as it currently stands would work as a flight reservation system, how many of you reading this now would like to see something like it on your desktop? Probably not many. It looks basic, it lacks polish, and it's not very user friendly at all.

That's what this tutorial's main focus is: improving the user experience in a Swing application, and turning a functional application into a marketable application. You'll learn how to give your applications the improvements they need to look and feel like a professional application. Swing offers the tools to do this with less effort than you may imagine, and there are many third-party applications, some of which I'll discuss in this tutorial, that fill the gaps that Swing doesn't address.

## Sample application enhancements

Here's a look at the areas of focus on in this tutorial, which will turn the basic application into a professional one:

- **JTable improvements.** Right now, the table that presents the results of a search to the user is pretty boring and not interactive at all. The results have no order, the data cannot be sorted, and the table cannot be

modified by the user to improve the view of it; in addition, it looks boring, with only white backgrounds, black text, and blue highlights. We can change all of that -- we can allow the user to sort columns, we can change how every cell looks, and we can change the table's preferences.

- **Thread issues.** Threading is one of the most tricky and difficult areas to understand in programming; unfortunately, with Swing, it doesn't get any easier. Fortunately, there are Sun classes not released with Swing that make working with threads in Swing much easier and eliminate many of the pitfalls that usually accompany threaded programming.
- **Custom components.** Because the usual widgets that accompany Swing are sometimes just plain boring, or because they might not function in the way we'd like, we can create custom components that will do exactly what we want them to. In this application, that Purchase button is just plain boring looking -- it's an important button and should reflect that fact. It will when we're done with it.
- **Custom look and feel.** Perhaps the coolest thing about Swing is its ability to change the look and feel of an entire application without affecting how the application itself functions. While it is incredibly easy to change the look and feel of an application, it is incredibly difficult to create a new look and feel from scratch. Because we want our flight reservation system to have a look and feel all its own, we'll create a simple example of a new look and feel to show how it can be done.

---

## Section 3. Intermediate JTable

### Introduction

If you've ever worked with the JTable before, you've surely encountered some of the difficult concepts it brings into play. Probably the most common complaint about the JTable from UI developers is that the basic version is just too simple, and is of no use in a real application. However, a more advanced JTable, one that becomes useful, adds many layers of complexity that not all developers want to deal with. Well, they've got no choice.

First, let's think about what's insufficient in the flight reservation JTable. At first glance, you can see that the colors that have chosen for the cells, or for highlighting, might not be what you desire. Or perhaps you don't want those gridlines showing up. So, that's one thing you can change: the *look* of it.

What about the *feel*, though? What else is wrong with the basic `JTable`? Well, for one, it doesn't provide any sorting mechanism. That's a big gap -- a `JTable` presents data to the user, and it should let the user decide how to arrange that data once it's on the screen.

What about some other little tidbits? How about text alignment? Users are often accustomed to having text left-aligned and numbers right-aligned. What about editing the contents of the cell? Some users expect that kind of behavior as well.

OK, OK, that's enough `JTable`-bashing. Instead of talking about how lacking a simple `JTable` is, let me show you how to utilize its potential and turn the plain `JTable` in the flight reservation system into a nicer table that users can tweak to improve their UI experience.

## JTable properties

The easiest way to start changing the `JTable` is to start changing its *properties*. The `JTable` class offers many, many functions that allow you to quickly tailor its look and feel without doing any overly difficult coding. As you've probably figured out by now, these functions for changing the properties are good for simple instances, but aren't sufficient for more complex needs (you'll find this a recurring theme throughout Swing). These functions include:

- **`setAutoCreateColumnsFromModel()`**: This function allows you to tell the table to automatically create the columns from the `TableModel`; in common practice, it should be set to `true`.
- **`setAutoResizeMode()`**: This changes the behavior when the user resizes the columns in the `JTable`. There are five possible values, and each value changes how the other columns' sizes change when the user resizes one column:
  - **`AUTO_RESIZE_OFF`**: The other columns don't change at all.
  - **`AUTO_RESIZE_NEXT_COLUMN`**: Only the size of the next column is changed.
  - **`AUTO_RESIZE_SUBSEQUENT_COLUMNS`**: The size of every column after the one being resized is changed.
  - **`AUTO_RESIZE_LAST_COLUMN`**: Only the last column is resized, ensuring that the columns always take up exactly the same amount of space as the table itself and will thus not require horizontal scrolling
  - **`AUTO_RESIZE_ALL_COLUMNS`**: All the columns are changed equally as the user-selected column is resized
- **`setCellSelectionEnabled()`**: Setting this to `true` allows the user to select a single cell; under the default behavior, the user would select a row.

- **setColumnSelectionAllowed()**: If this is set to `true`, when a user clicks on a cell, that cell's entire column will be selected
- **setGridColor()**: Changes the grid color of the table.
- **setInterCellSpacing()**: Changes the spacing between each cell, and thus the size of the grid lines.
- **setRowHeight()**: Changes the row height of the table.
- **setRowSelectionAllowed()**: If this is set to `true`, when a user clicks on a cell, the entire row containing that cell is selected.
- **setSelectionBackground()**: Changes the background color of a selected cell.
- **setSelectionForeground()**: Changes the foreground color of a selected cell.
- **setBackground()**: Changes the background color of a non-selected cell.
- **setForeground()**: Changes the foreground color of a non-selected cell.
- **setShowGrid()**: Allows you to hide the grid entirely.

As you can see, there's plenty you can do to a `JTable` without much effort. These properties allow you to tailor the `JTable` to your own needs in an application. However, as nice as these little properties are, they don't address some of the more pressing needs of the example application.

## TableRenderer

In the previous section, you saw that you could change the foreground color and background color of a cell when it is selected and unselected. That's a nice feature, but it's limited: What if you want to change the font as well? What if you want to change the color depending on the specific value of that cell?

This more advanced drawing of a table cell is handled by an interface called a `TableRenderer`. By creating a class that implements this interface, you can create a custom palette to do any kind of painting you need to do. The `JTable` will then pass all painting duties to this new class.

For the example application, we'll change the way the table looks to match the following rules:

- Give unselected rows a white background and a black plain text foreground.
- Give selected rows a green background and a black text foreground.

- Give rows that contain sold-out flights a dark gray background and a white text foreground.

The first step to using the new `TableRenderer` is to tell the `JTable` to use it instead of its custom built-in renderer. (The `JTable` has a default table cell renderer, which we will be overriding to do our own painting.) By attaching the renderer to `Object.class`, we are in effect telling the `JTable` to use our custom renderer for every cell.

```
getTblFlights().setDefaultRenderer(Object.class, new FlightTableRenderer());
```

Now that that step is quickly out of the way, let's look at the `FlightTableRenderer` itself. The `FlightTableRenderer` takes advantage of a built-in Swing class, the `DefaultTableCellRenderer`, which provides a good starting point. We can extend and override this class's only method to do our own painting. The actual painting logic should be self-explanatory.

```
public class FlightTableRenderer extends DefaultTableCellRenderer
{
    public Component getTableCellRendererComponent(JTable table,
        Object value,
        boolean isSelected,
        boolean hasFocus,
        int row,
        int column)
    {
        setText(value.toString());

        if (((Integer)table.getValueAt(row, 3)).intValue() < 1)
        {
            setBackground(Color.GRAY);
            setForeground(Color.WHITE);
        }
        else
        {
            if (isSelected)
            {
                setBackground(Color.GREEN);
                setForeground(Color.BLACK);
            }
            else
            {
                setBackground(Color.WHITE);
                setForeground(Color.BLACK);
            }
        }
        return this;
    }
}
```

After inserting this code into the example application, the table of results looks drastically different. It also conveys more information to the user, as it grays out flights that are already sold out:

Flight Reservation

Departure: AUS      Destination: DFW      Search

Departure	Destination	Flight Num	Number of Tix
AUS	DFW	270	36
AUS	DFW	8369	0
AUS	DFW	9911	32

Number of Tickets: 0      Purchase

## TableModel

In the introductory tutorial, you learned how to use `TableModels` -- and, specifically, how to use them to more easily manage your data. However, besides handling the way in which data is presented in the table, the `TableModel` can also change the behavior of the table in two distinct ways.

- The `TableModel` tells the `JTable` how to align the data in each column. What exactly does this mean? Well, for example, `Strings` are typically left aligned, and numbers and dates are typically right aligned. Numbers representing currency amounts are typically aligned by the decimal point. `TableModel` contains a method called `getColumnClass()` that handles all the alignment issues. Fortunately, the `JTable` has numerous built-in `TableRenderers` that know how to handle certain classes, including `Integers`, `Strings`, and `Doubles`. You would change the alignment by providing the following function in your table model:

```
public Class getColumnClass(int col)
{
    if (col == 2 || col == 3)
        return Integer.class;
    else
        return String.class;
}
```

Unfortunately, this call conflicts with the code we created for the `FlightCellRenderer` and attached to `Object.class`. This is just a case of too many examples for the limited amount of code that we have, so you'll have to believe me that this works, and that the `JTable` knows how to align certain classes. With the existing `FlightCellRenderer` in



our example, to get our data aligned correctly, we'd need to modify our code slightly, and add lines for `setHorizontalAlignment()` to align the data in the way that we'd like.

```
if (column == 2 || column == 3)
    setHorizontalAlignment(SwingConstants.RIGHT);
else
    setHorizontalAlignment(SwingConstants.LEFT);
```

- The `TableModel` allows individual cells to be edited. Much like default cell renderers, the `JTable` provides default cell editors for some built-in classes, like `String` and `Integer`. In other words, it knows how to provide editing capabilities for these built-in classes. The `TableModel` only needs to tell the `JTable` which cells are allowed to be edited and how to set the values once the editing is complete, and the `JTable` will handle the rest.

```
public boolean isCellEditable(int row, int col)
{
    return col == 3;
}

public void setValueAt(Object value, int row, int col)
{
    if (col == 3)
        ((Flight)data.get(row)).setRemainingTixx(new Integer(value.toString()).intValue());
}
```

Now, you may ask, "What if I don't want to use one of Swing's built-in editors?" Well, my initial answer would be, "That's a shame." Creating a custom editor -- one to edit dates cleanly, for instance -- is very difficult. As you've seen in other areas, the simplicity of letting Swing handle the editing of built-in classes is offset by the complexity of creating new classes for editing classes that aren't built into the `JTable`. The issue of creating a new class to edit cells is beyond the scope of this tutorial, and if you decide you absolutely need this feature, don't say I didn't warn you. For those of you interested in learning about how to create a custom cell editor, check out [Resources](#) for a site that offers a how-to.

## Sorting

Users of tables -- whether they're tables in a desktop application or a Web application -- have come to expect the ability to sort the data in those tables by column. In other words, they expect to be able to click on the column heading and see the data rearranged to match either an ascending or descending order indexed on the column that was clicked.

Unfortunately, the `JTable` doesn't have this feature built into it. This is a real shame because nearly every user expects it. This feature is already built into some potential future versions of Swing; but because so many people have come to expect it, it's already offered in a third-party addition to the `JTable`, and is fairly easy to add to our flight reservation system. The ironic thing is that the "third-party application" that most people use to sort their tables is provided by Sun itself in its `JTable`

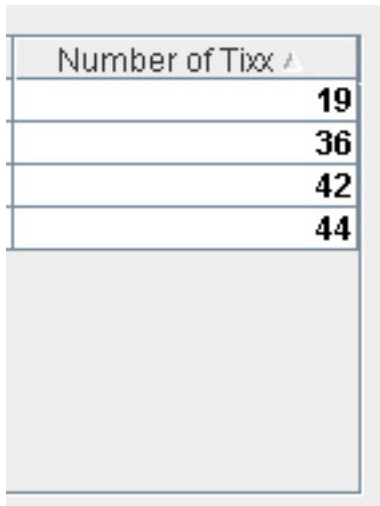
documentation. Makes you wonder why Sun doesn't just include it in Swing automatically, instead of making people pore through documentation to find it, doesn't it?

The class that Sun provides is called `TableSorter`, and it is incredibly easy to use in any `JTable`. The `TableSorter` class itself handles everything automatically, sorting the data in either ascending or descending order depending on the mouse button pressed, and also using an arrow to denote the direction of the sort.

With only two lines of code, we can add sorting abilities to the flight reservation system:

```
TableSorter sorter = new TableSorter(flightModel, getTblFlights().getTableHeader());
getTblFlights().setModel(sorter);
```

The image below is what the sorting abilities look like in our application. Notice the arrow after the word "Tixx." It indicates that the column is sorted, and also indicates the direction of the sort -- either ascending or descending:

A screenshot of a Java Swing window containing a JTable. The table has a single column with a header cell containing the text "Number of Tixx" followed by a small upward-pointing arrow. Below the header are four data cells containing the numbers 19, 36, 42, and 44, respectively. The numbers are in descending order, indicating that the table is sorted in descending order. The table is enclosed in a light gray border.

Number of Tixx ^
19
36
42
44

## JTable summary

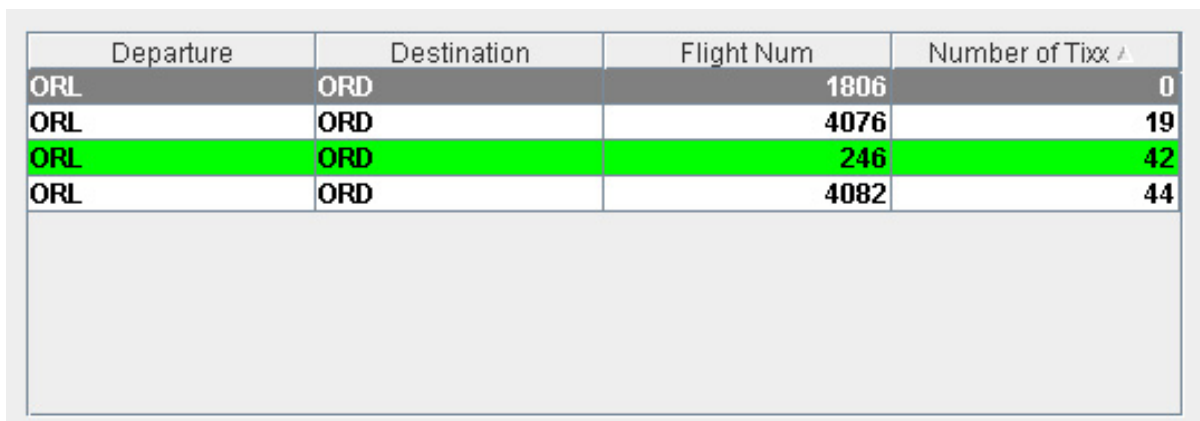
So, in just a few sections and with some new classes, we've managed to transform our `JTable` from a stale and boring table into a ... somewhat less boring table (let's not kid ourselves). In any case, we have added much to our `JTable` through some simple lessons, and you should be able to add even more to it by taking these lessons and building on them. Here's a quick review what we covered:

- You can call set methods on the `JTable` to quickly change some simple properties of the table itself. Some of these properties are the grid color, the grid size, the row height, and other "sugar-coating" items in the `JTable`.
- You can change how each cell looks by creating a new `TableRenderer`. By creating a `TableRenderer`, we are also taking all responsibility for painting the cell, in effect telling the `JTable` that we will handle all the

painting. Keep this in mind when you create your own `TableRenderer` -- any errors will become evident quickly. Also, by creating your own `TableRenderer`, you take responsibility for the alignment of the data, which can otherwise be handled by the `TableModel`.

- You can use the `TableModel` to change the way in which the data is aligned in the column (provided you haven't overridden the `TableRenderer`) and also to allow for editing in the cells themselves.
- The `JTable` handles many types of data automatically, including `Strings`, `Integers`, and `Doubles`. You can take advantage of this fact by using the built-in renderers and editors for these classes. Unfortunately, for those classes that aren't automatically supplied by the `JTable`, you have to provide your own renderer (which we learned about here) and editor (which is complex and beyond the scope of this tutorial).
- Finally, you learned how to add sorting abilities to your `JTables` by using Sun's `TableSorter` class. Although Sun provides this class with the Swing documentation, it is not a supported Swing class, and needs to be shipped with any classes you create. In future versions of Swing, sorting will be built in.

The image below shows how the `JTable` in the flight reservation system looks now:



Departure	Destination	Flight Num	Number of Tickets
ORL	ORD	1806	0
ORL	ORD	4076	19
ORL	ORD	246	42
ORL	ORD	4082	44

---

## Section 4. Thread-safe Swing

### When the UI freezes

As I mention the word *threading*, I think I hear a collective groan coming from the crowd. Yes, even Swing isn't immune from threading issues, and if you're like me, that's not something you wanted to hear. Threading issues are often difficult things to code around -- the concepts are abstract, and when errors occur, they're difficult to test and hard to fix.

In Swing, the concepts themselves at least are a lot easier to understand and code for. For example, in our flight reservation system, suppose the user clicks on the button to purchase the tickets -- in the current version of the application, this process completes quickly because we are using a fake database. But suppose the app were talking to a real database located somewhere else, as in a classic client-server application. When the user clicks the Purchase button, we need to call and update the database, and wait for it to return. Are you really going to make your user wait that entire time, possibly up to a minute, while you process this purchase? Hopefully not!

There are multiple problems that occur in Swing when someone codes an action like this. What the user will notice first is that he or she is now locked out of the application, and can not interact with it any more; Swing operates on a single thread (the event-dispatch thread), meaning that as the thread sits and waits on the database call to complete, any other user interaction (trying to select another destination city, for example) is thrown on the queue for the thread to complete, and thus is left waiting for the call to the database to complete. We can't lock out our users from their own application!

Another annoying feature of Swing is that when the Swing thread is busy and the application needs to repaint itself (if you minimize and then maximize the frame, for example), the repaint command is also blocked by the database call, and the user ends up seeing a gray rectangle. This problem will be fixed in the next version of Swing, but we have to deal with it in the meantime.

These two problems alone should show you how important it is to deal with threading issues in Swing correctly -- you can't possibly call your application professional if you lock your users out of it and turn it into a gray rectangle every time the application needs to run some code that takes longer than a few milliseconds. Luckily, there are solutions to these common problems.

## Working on the Event-Dispatch Thread

Swing contains a method called `SwingUtilities.invokeLater()` that actually deals with a problem that is in some ways a mirror image of the one we considered in the previous section. Instead of finding a solution for performing time-intensive operations on a separate thread from the event-dispatch thread, this function allows non-UI related threads to perform tasks on the event-dispatch thread. Why on earth would anyone want to do this? Well, the easiest example to understand is the startup of a GUI application. Think about what an application typically does during startup: open server connections, load preferences, build the GUI, etc. This is the perfect opportunity to delegate the building of the GUI to the event-dispatch thread, letting the main application thread continue doing other tasks as the event-dispatch thread takes care of the Swing-related issues. This solution improves application start-up performance.

We originally started our flight reservation system like this:

```
public static void main(String[] args)
{
```

```

FlightReservation f = new FlightReservation();
f.setVisible(true);
}

```

However, we should take advantage of the performance improvements offered by the `SwingUtilities.invokeLater()` method and allow the event-dispatch thread to build the Swing components (granted, in this small application, the performance improvement is imperceptible, so you'll have to use your imagination here).

```

public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        {
            FlightReservation f = new FlightReservation();
            f.setVisible(true);
        }
    });
    // possibly open files, get DB connections here
}

```

However, this still doesn't fix our original problem: moving slow processes off of the event-dispatch thread. We'll tackle that next.

## Third-party Swing threading solutions

In an ironic twist (along the lines of one you've seen already), there is a solution to the problem I posed in the past two sections, and it's published in Sun's Swing documentation -- although, like the `TableSorter`, it's not published as part of the Java release. Why Sun does this, I'm not sure. Perhaps it's to make sure that you to read the documentation. In any case, the supplied solution from Sun solves the problem of running time-consuming processes on the event-dispatch thread.

The class is called `SwingWorker`, and at its foundation, it works like any other thread. However, it is specifically tailored to work with Swing apps, by clearly separating the logic that should be performed off of the event-dispatch thread from the logic that should be performed on the event-dispatch thread but is dependent on the results of the logic that isn't on the event-dispatch thread. Confusing? It shouldn't be, once you see it in action.

Let's look at what happens when the user presses the Search button, and break down the parts of the call and decide what threads they should operate on.

```

// should occur on the event-dispatch thread as it deals with Swing
final String dest = getComboDest().getSelectedItem().toString();
final String depart = getComboDepart().getSelectedItem().toString();

// should NOT occur on the event-dispatch thread, as it could be time consuming
List l = DataHandler.searchRecords(depart, dest);

// should occur on the event-dispatch thread, but is dependent on the results from
the previous line
flightModel.updateData(l);

```

The `SwingWorker` class requires that all code that *shouldn't* be performed on the event-dispatch thread go in its `construct()` method. The class also requires that

all code that *should* go on the event-dispatch thread, but is dependent on the results of the `construct()` method, go in the `finished()` method. The key link between the two methods is that the `construct()` method returns an `Object` that the `finished()` method can access by calling `get()`. With that in mind, we can modify the above code, implementing the `SwingWorker` to remove the blocking code from the event-dispatch thread, and ensuring that our application won't prevent the user from interacting with it.

```
final String dest = getComboDest().getSelectedItem().toString();
final String depart = getComboDepart().getSelectedItem().toString();
SwingWorker worker = new SwingWorker()
{
    public Object construct()
    {
        List l = DataHandler.searchRecords(depart, dest);
        return l;
    }

    public void finished()
    {
        flightModel.updateData((List)get());
    }
};
worker.start();
```

---

## Section 5. Custom components

### Introduction

Have you ever had that feeling that sometimes what you have just isn't enough? Do you think that Swing provides everything you could possibly need to make your UI look exactly as you want it to look? Chances are, your answer is *yes* to the first question and *no* to the second. Although Swing provides dozens of components and widgets, creative people out there are always demanding more.

Luckily, Swing provides the necessary tools to remedy this problem. You want a new component? No problem. You can whip one up with only a few lines of code. Want to change the behavior of an already existing component, to tailor it to the way you want it to look in your own application? That can be easily done as well. By using Java's class hierarchy, modifying an existing Swing widget to create a custom component is as simple as subclassing the existing widget. If you want to create a completely new custom component, it's like building blocks -- just combine the existing widgets into one bigger widget.

In this section, we'll create custom components in both of these ways, starting with the modification of an existing Swing component.

### Modifying an existing component

The first step to modifying an existing component is, naturally, to decide what the heck you're going to change about it. For the flight reservation system, in this example, we'll modify the Purchase button, because it's just too important to look like other buttons in the application. We'll create something called a `JCoolButton`, which will hide the border of the button until the mouse is over it; at that point, it will display the border.

The first step in modifying a component is to subclass it -- in this case, we'll be subclassing the `JButton`. By subclassing the constructor and pointing to a new function (called `init()` in this example), you can create your own behavior different from the default component. An important point to remember is that when you subclass a Swing widget and you intend to let others reuse it, you must subclass every constructor to ensure that any user will get your desired behavior.

Take a look at some example code. You'll see how the `JCoolButton` modifies the `JButton` to only paint the border when the mouse is over it, creating a hover effect on the button.

```
public JCoolButton()
{
    super();
    init();
}

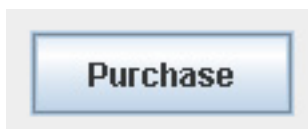
private void init()
{
    setBorderPainted(false);
    addMouseListener(new MouseAdapter()
    {
        public void mouseEntered(MouseEvent arg0)
        {
            setBorderPainted(true);
        }

        public void mouseExited(MouseEvent arg0)
        {
            setBorderPainted(false);
        }
    });
}
```

Here's the button without the mouseover:



And here it is with the mouseover:



## More modifications to `JCoolButton`

You may notice the `JCoolButton` isn't quite perfect yet, because the gradient



background is still painted when the mouse isn't over it, and that makes it appear awkward. We need to add some more code to make the `JCoolButton` completely blend into the background `JPanel` when there is no mouse over it, enhancing the cool effect of the button.

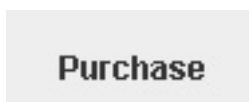
This brings us to an important lesson in how Swing components work. It's very important to understand how a component actually paints itself, in what order, and what the ramifications are of overriding the individual functions. This lesson begins with the `paint()` function in `JComponent`. Because it is in `JComponent`, every Swing component contains it, and it is, in fact, the function that is responsible for painting every single component in Swing.

The `paint()` documentation tells us that when the method is called, it in turn calls three other methods in `JComponent`: first `paintComponent()`, then `paintBorder()`, and finally `paintChildren()`. From the order in which the methods are called, you can see that Swing paints itself from the bottom to the top. The important lesson to learn, though, is that in order to override how a component looks, you need to override the `paintComponent()` method, which is directly responsible for painting how an individual component appears on the screen.

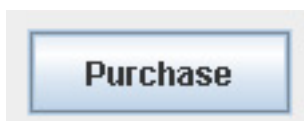
In our `JCoolButton`, let's fix the problem where the blue gradient background is painted even when the mouse is not over the button. We want it to blend right into the background instead. We can accomplish this by overriding the `paintComponent()` method and keeping track of when the mouse is over the button and when it isn't, and painting the button for each state. One important note before we look at the code: when you override the `paintComponent()` method, you are responsible for *all* painting. In other words, you're responsible for the text, the background, the decoration -- basically everything but the border.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    if (!mouseOver)
    {
        g.setColor(getParent().getBackground());
        g.fillRect(0,0,getSize().width, getSize().height);
        g.setColor(getForeground());
        int width = SwingUtilities.computeStringWidth(getFontMetrics(getFont()), getText());
        int height = getFontMetrics(getFont()).getHeight();
        g.drawString(getText(), getWidth()/2 - width/2, getHeight()/2 + height/2);
    }
}
```

Here's our newly modified button without the mouseover:



And here it is with the mouseover:





## Building a completely new component

You creative types may feel constrained by the existing Swing components -- not only by the way they look, but by how they work. You may think that there are no existing components that can do what you want a widget to do. You have a vision, and you want it in your UI.

Swing gives you the ability to realize your dreams, to let your creative impulses run wild, and to create any component with any functionality you desire -- and it takes only four steps:

1. Identify the existing Swing widgets you need to combine to make your new widget. The existing Swing components provide building blocks that should be used for the new widgets. Their advantage is obvious -- they're complete components that have been extensively tested.
2. Combine the widgets together visually to make the component look how you want it to look. The best way to do this is to add the chosen components to a JPanel, being sure to use a layout manager (because you're not sure how big or small other users may want it).
3. Make the components interact together properly to get the desired behavior from the components. This would involve creating private functions to handle events, route data from one part to the other, have the whole thing paint itself properly -- basically, anything you need to do to make it behave like you want it to.
4. Wrap a public API around the finished component so that others can understand how to use it easily.

## The JMenuButton

OK, enough with the abstract descriptions. Let's get to some examples! We'll create a new component and go through the four steps outlined in the previous section to show you how you can create a brand new component.

The component we'll be creating is a JMenuButton, a new component that acts like a normal JButton, but with additional functionality: The user can press an arrow button on the side of the JButton to reveal additional choices. If this description is confusing (as I'm sure it is), think about the Back button on your Web browser: clicking it sends you back to the last Web page you were reading, but clicking and holding it pops up a list of the pages in your recent browsing history. This is essentially a JMenuButton, and we'll be building one of these in Swing. You'll notice that something like this doesn't exist in Swing currently, but all the building blocks are there to build it.

Here's a look at the finished product, the goal we are aiming for:



One further note: We won't be implementing this component in the flight reservation system as there's no appropriate place for it. Nonetheless, feel free to use it in your own applications as you see fit.

## Step 1: Choose components

We'll need four components to create the JMenuButton:

- A JButton that serves as the main button (the Back portion of the component). This serves as the primary choice in the component.
- A JButton that serves as the arrow button. The arrow button displays the pop-up menu that contains the alternative choices in the component.
- A JPopupMenu that will be displayed whenever the arrow button is pressed and the alternative choices are displayed.
- JMenuItem items that will go into the JPopupMenu and serve as the alternative choices.

## Step 2: Lay out the components

As I pointed out in [Building a completely new component](#), the easiest way to create a new component is to add existing components to a JPanel. In our example, we will use a JPanel as the base of the component, and place the two buttons on top of it. In fact, our JMenuButton subclasses from JPanel, and not any of the components that it contains. Choosing the layout for our components on the JPanel is straightforward enough, and we can just use a BorderLayout.

```
this.setLayout(new java.awt.BorderLayout());
this.add(getBtnMain(), java.awt.BorderLayout.CENTER);
this.add(getBtnArrow(), java.awt.BorderLayout.EAST);
```

## Step 3: Internal interaction

The third step in creating a new component is to get it working properly (obviously). In this example, we need to get the arrow button to launch the pop-up menu whenever it is pressed. Note that the main button doesn't necessarily do anything internally. Externally, it needs to be listened to, but internally, it doesn't change the state or appearance of the component, and thus can be ignored in this step. Also,

the individual JMenuItem items in the JPopupMenu can be ignored internally as well -- we don't even have any added when we start, and they don't change the component, either.

Here's the code that will handle the internal interaction of the components:

```
getBtnArrow().addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        getPopup().show(getBtnMain(), 0, getHeight());
    }
});
```

## Step 4: Create a public API

The final step is perhaps the most important in creating a new component -- you must wrap the component in an easy-to-use public API, so that other people who want to use your component can do so easily. After all, what's the point of creating something new if you don't let anyone else use it? Creative genius should be shared, after all.

Any time you are creating a public API wrapper around your component, you should aim to keep the format the same as existing Swing components -- use get/set methods, and try not to change any behavior that UI developers would expect from a Swing component (for example, don't change any methods in JComponent that UI developers would expect to act the same from component to component).

There are two areas that we need to be concerned with in our JMenuItemButton. First, we need to be able to add alternative choices to the JMenuItemButton. We can create a new method to handle this:

```
public void add(JMenuItem item)
{
    getPopup().add(item);
}
```

Second, we need to be able to handle action events that the main button and the alternative choices could trigger -- after all, users need to know when a button is pressed or an alternative choice is selected.

```
public void addActionListener(ActionListener l)
{
    getBtnMain().addActionListener(l);
    for (int i=0; i<getPopup().getSubElements().length; i++)
    {
        JMenuItem e = (JMenuItem)getPopup().getSubElements()[i];
        e.addActionListener(l);
    }
}

public void removeActionListener(ActionListener l)
{
    getBtnMain().removeActionListener(l);
    for (int i=0; i<getPopup().getSubElements().length; i++)
    {
        JMenuItem e = (JMenuItem)getPopup().getSubElements()[i];
        e.removeActionListener(l);
    }
}
```

```
}
```

---

That's it. That's the end of the process to create `JMenuButton`. After only a few sections, we've managed to create a new and functional Swing component. By following the four steps outlined in this section, you too can create your own component.

The `JMenuButton` isn't meant to be a professional and polished component -- it merely exists to serve as an example. Thus, if you choose to use it in your own applications, I'd suggest additional public methods and further testing.

---

## Section 6. Custom look and feel

### Background

One of the coolest features of Swing is its ability to easily change the look and feel of an application. The look and feel of an application is a combination of two things (and remember these terms, as I'll refer to them throughout this section):

- The *look*, which is how the components appear visually: what colors they use, what fonts they use, shading, etc.
- The *feel*, which is how the components interact with users: how they react to a right-click, how they react to mouse drags, etc.

The look and feel of an application is governed by the Swing class `javax.swing.LookAndFeel`; we'll refer to an instance of this generically as a *LookAndFeel*. There's a small but distinct difference between the look and the feel, one that we'll examine in detail when we talk about [Synth](#).

Another ironic thing about how custom `LookAndFeels` work in Swing: the concepts and code needed to create a new `javax.swing.LookAndFeel` class are complex (and beyond the scope of this tutorial), but once the `javax.swing.LookAndFeel` is done and packaged, it is incredibly easy to make your application use it.

In fact, you can change the look and feel of your entire application by adding only one line of code!

```
UIManager.setLookAndFeel(new WindowsLookAndFeel());
```

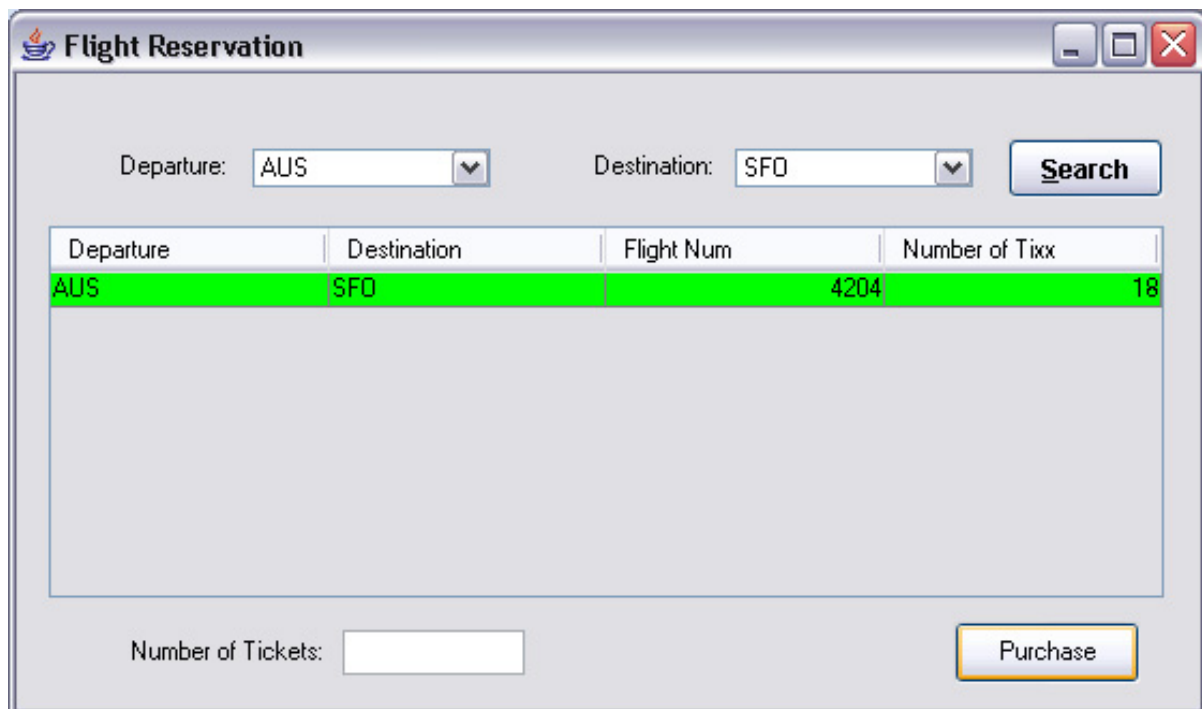
### Swing's LookAndFeels

Swing comes installed with multiple `LookAndFeels` pre-installed. These `LookAndFeels` match up to the common OSes on the market right now, but have an unfortunate side-effect: the `LookAndFeel` that resembles Windows XP is only

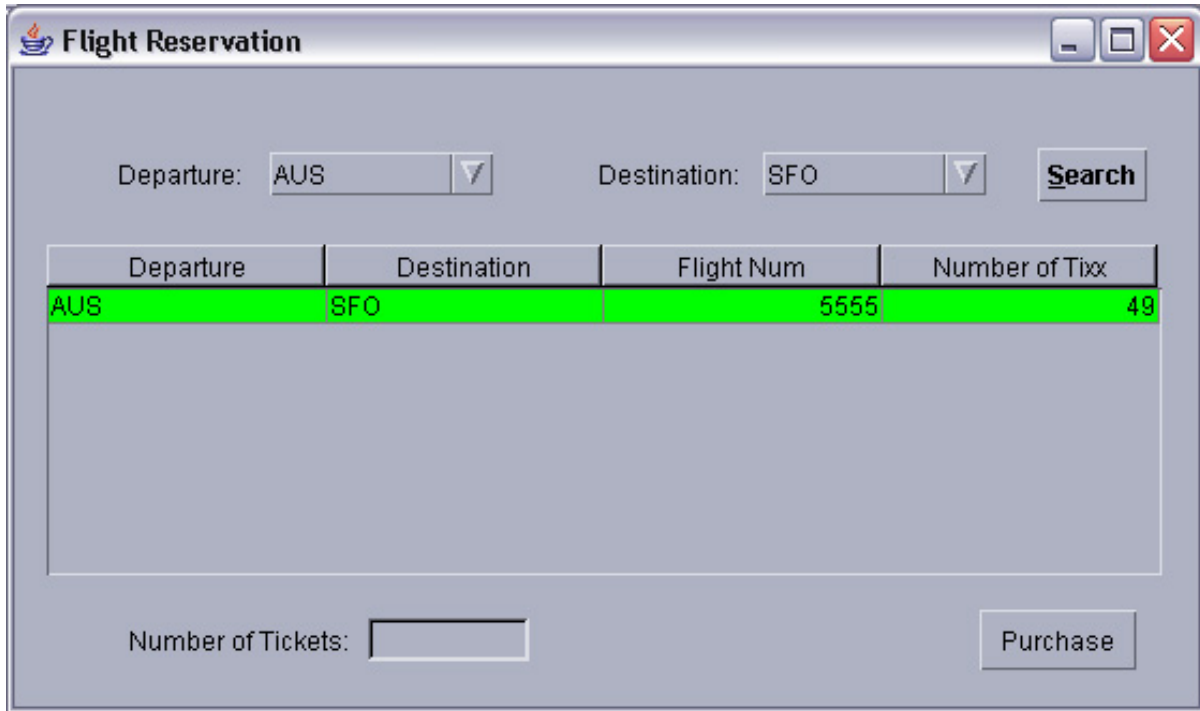
available on Windows, the Macintosh LookAndFeel is only available on Mac OS, and the GTK LookAndFeel is only available on Linux systems. Sun has also made the Ocean LookAndFeel, an attempt to provide a good-looking cross-platform look and feel.

Let's take a look at how our flight reservation system looks in the different installed LookAndFeels (though these will not include Mac or GTK, because I'm writing this tutorial on a Windows machine). Remember, only one line of code needs to be changed to completely change how our updated flight reservation system application looks.

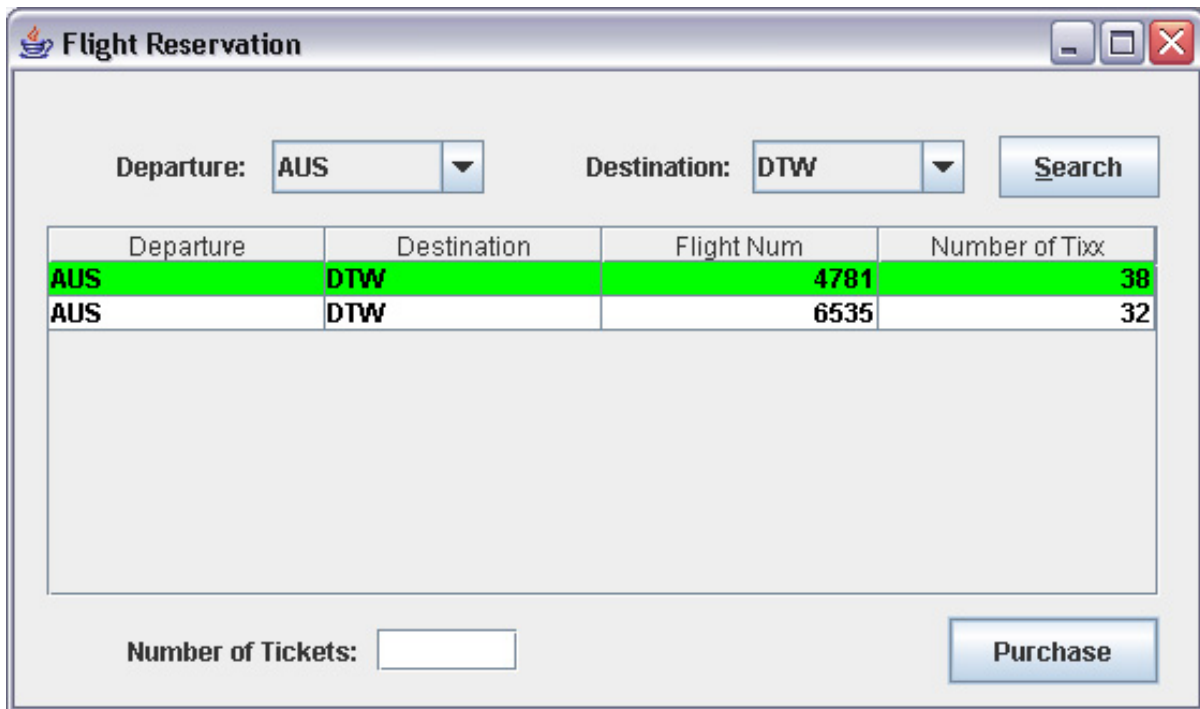
Here's the Windows LookAndFeel:



Here's the Motif LookAndFeel:



And here's the Ocean LookAndFeel:



## UIManager

The easiest way to start changing the look and feel of your apps is to learn how to use the UIManager. The UIManager provides access to everything that has to do with the installed look and feel -- and by everything, I mean *everything*. Every possible color, every possible font, and every possible border can be changed and

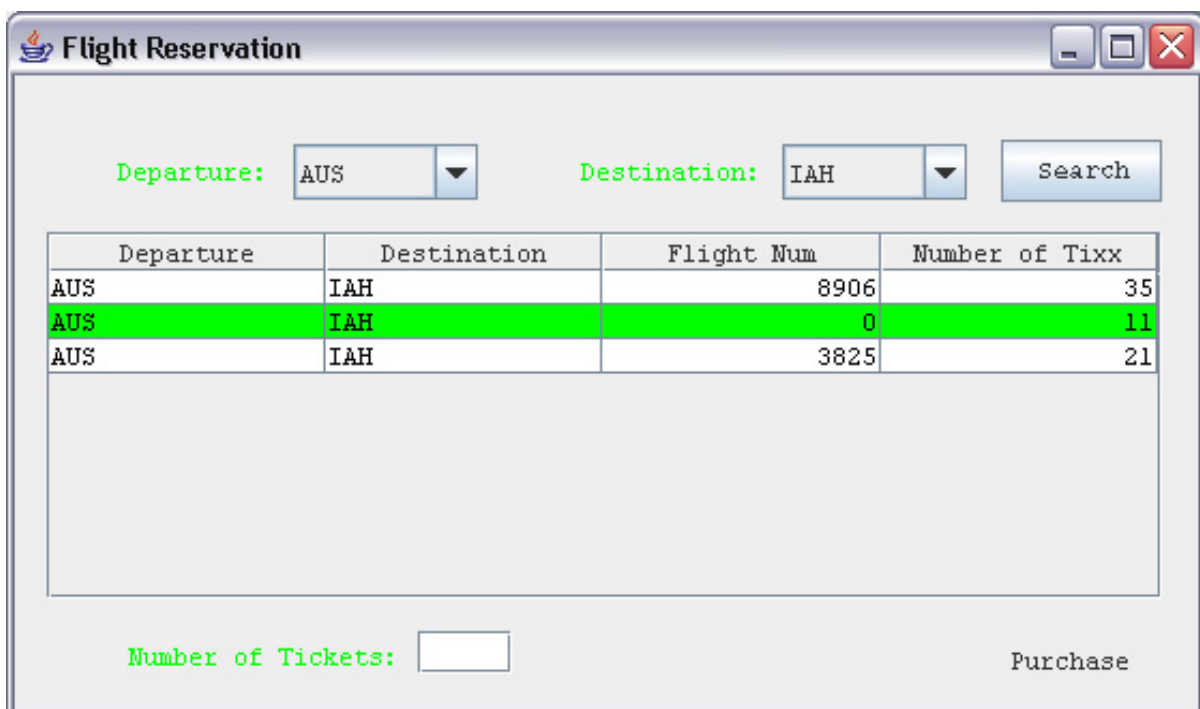
manipulated by the UIManager. The UIManager acts as a `HashTable` that contains all these values, and ties them to `Strings` that act as the key in the hashtable relationship.

Therein lies the difficult part of working with the UIManager -- these `Strings` that act as the keys are not documented anywhere on the Sun site. Fortunately, by examining the installed `LookAndFeels` that come shipped with Swing, you can see what keys Sun used and use them yourself. It's unfortunate that these keys aren't documented anywhere, as this only adds to the complexity of creating a new custom look and feel.

We'll use the UIManager to change the colors for the labels from blue to green and the font that the entire application uses:

```
Font font = new Font("Courier", Font.PLAIN, 12);
UIManager.put("Button.font", font);
UIManager.put("Table.font", font);
UIManager.put("Label.font", font);
UIManager.put("ComboBox.font", font);
UIManager.put("TextField.font", font);
UIManager.put("TableHeader.font", font);
UIManager.put("Label.foreground", Color.GREEN);
```

Here's our newly tweaked application:



## Packing it all into a LookAndFeel

Of course, this type of coding isn't conducive to object-oriented programming, as you don't want to retype these lines of code in every application that needs to have this look and feel. You also can't easily let others use your new creation this way.

Sun's solution is to provide the `javax.swing.LookAndFeel` class, which lets you

easily package up all the information needed to create a look and feel for your application. It also lets you describe how all the pieces come together to create the look and feel you want to distribute to everyone.

There are two ways to create a `LookAndFeel`. One is to subclass the `javax.swing.LookAndFeel` class itself, though this is the more difficult option. The better solution is to subclass one of the existing `LookAndFeels` provided by Swing -- either the `javax.swing.plaf.metal.MetalLookAndFeel` or the `javax.swing.plaf.basic.BasicLookAndFeel` (a building block look and feel that doesn't have any visuals but is provided to serve as a basis for others to build on).

There's also some basic information that every `LookAndFeel` needs to contain -- things that tell Swing about the `LookAndFeel` and let others know about it as well, should you decide to package it up and distribute it.

- **`getDescription()`**: A description of the look and feel.
- **`getID()`**: A unique ID that can be used to identify the look and feel.
- **`getName()`**: The name of the look and feel.
- **`isNativeLookAndFeel()`**: Indicates whether this look and feel is native to the OS; any custom look and feel should return `true` from this function.
- **`isSupportedLookAndFeel()`**: Should also return `true` for any custom look and feel.

Let's look at this code in our new custom `LookAndFeel` class, the `FlightLookAndFeel`.

```
public String getDescription()
{
    return "The Flight Look And Feel is for the Intermediate Swing tutorial";
}

public String getID()
{
    return "FlightLookAndFeel 1.0";
}

public String getName()
{
    return "FlightLookAndFeel";
}

public boolean isNativeLookAndFeel()
{
    return true;
}

public boolean isSupportedLookAndFeel()
{
    return true;
}
```

## More on the `FlightLookAndFeel`



The final step in creating the `FlightLookAndFeel` class is to actually describe what the look and feel should change. In the example discussed in [UIManager](#), we were merely overriding certain colors and fonts that were already in the `MetalLookAndFeel`. We are going to reuse this code to create our look and feel.

There's a class that functions like the `UIManager` called the `UIDefaults`. The difference between these is subtle -- the `UIManager` should be used outside of the `LookAndFeel` classes, as it represents all the look and feel values after it is loaded. The `UIDefaults` represents these same values as they are being loaded.

The `LookAndFeel` class has a `getDefaults()` method that loads up these values, and then uses them to create the look and feel. In order to get our own values in there instead of the `Metal` ones we are overriding, we must first call the `MetalLookAndFeel`'s `getDefaults()` function to load every value that we are not overriding (if we didn't do this, we'd end up having a weird-looking look and feel, missing colors, sizes, and so on). After letting the `MetalLookAndFeel` have the first stab at creating everything it needs, we can just override the values we want and create the `FlightLookAndFeel` the way we want it to look.

```
public UIDefaults getDefaults()
{
    UIDefaults def = super.getDefaults();
    Font font = new Font("Courier", Font.PLAIN, 12);
    def.put("Button.font", font);
    def.put("Table.font", font);
    def.put("Label.font", font);
    def.put("ComboBox.font", font);
    def.put("TextField.font", font);
    def.put("TableHeader.font", font);
    def.put("Label.foreground", Color.GREEN);
    return def;
}
```

## More on custom LookAndFeels

While the previous sections were sufficient to create a custom look and feel that dealt with color and fonts only, it really isn't a complete enough lesson in how to create a complete custom look and feel, one that would be comparable to the `WindowsLookAndFeel` or the `MotifLookAndFeel`. These go far beyond just modifying the colors and fonts; they change nearly everything from the default implementation -- how each component is drawn, how it reacts to mouse events, where components are placed when it has items added to it, and so on. Creating a complete look and feel is not an easy task.

This tutorial will not address the many steps need to create a custom look and feel. In fact, an entire tutorial could be created just to teach you how to create a custom look and feel; it is unfortunately that complex an undertaking. To give you a sense of things, the task requires the user to create a new class for every `Swing` component, outlining how it should look and feel. That's roughly 60 classes that you would need to create -- not something that can be done in an afternoon, and especially not something that could be summarized in a few sections in this tutorial.

As further proof as to the complexity of creating a complete custom look and feel,

the number of examples available on the Internet is amazingly small. Only about 20 to 30 commercial LookAndFeel are available for download -- and that's with Swing being on the market for seven years.

Fortunately, in J2SE 5.0, Swing introduced a new class that has made the process much easier and reduced the time needed to create a custom look and feel from about three months (as it required in 1.4) to three weeks (as it now requires). We'll discuss it in the next section.

## Synth

Synth is the newest LookAndFeel addition to Swing, but it's kind of a misnomer. It isn't really a LookAndFeel at all, which you'd find out if you tried to add it to your application: your application would turn completely white. Nor can you even modify the complete look and feel of an application once you begin using Synth. Synth allows you to only change the look of an application -- the borders, the colors, the fonts. It does not allow you to change how the application feels.

Synth in a nutshell is a skin that you install on your application. The skin contains information describing how to use external images and custom paint code to create the look, parsing this information from a single XML file that is loaded when the Synth LookAndFeel is installed in the application. The biggest advantage is the time savings it offers users -- instead of having to subclass 60 Java classes, you merely need to create one XML file and some graphics. The result of this advantage is a potential explosion in the number of custom LookAndFeels available for developers to choose from; this is Sun's ultimate goal in releasing this new class. In case you haven't been following the drama, Swing for years was criticized for being ugly. The Ocean LookAndFeel alleviated that complaint somewhat, and Synth offers the potential for creative developers to make some very nice LookAndFeels in the near future.

For a complete lesson on how to use Synth and how it will fit into your application, check out my [Advanced Synth](#) article published a few months ago (see [Resources](#) for a link), which will take you step-by-step through the process of creating a new custom look and feel using Synth.

---

## Section 7. Wrap-up

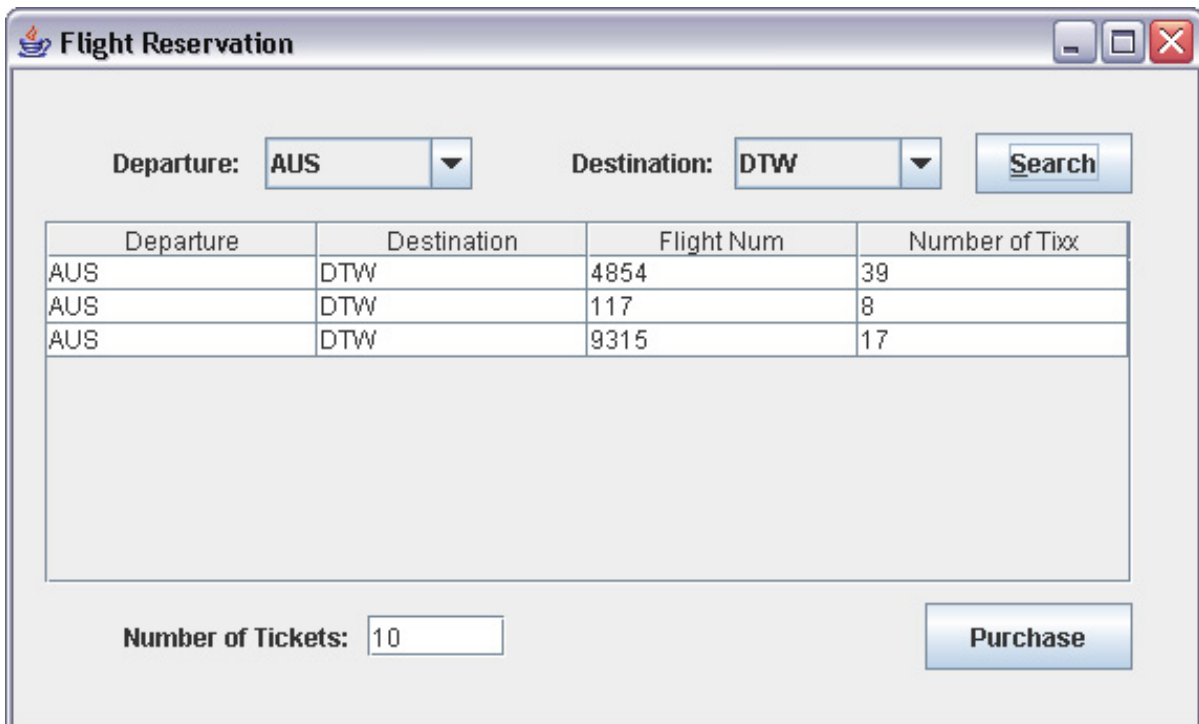
### Summary

This tutorial dealt with some intermediate-level issues that you may encounter when working with Swing when developing UI applications. It is meant to build upon the knowledge you gained in the [Introduction to Swing](#) tutorial, or to build on your

already existing Swing knowledge. It dealt with some common issues that UI developers come across in their applications. I've attempted to pick out a combination of the most interesting and most important issues.

As I've discussed many times throughout both tutorials, these tutorials are by no means exhaustive in their coverage of Swing -- there's just far too much to learn to be summarized in two tutorials. Hopefully what you've learned in this tutorial will not only improve your knowledge in a few fields of Swing, but pique your interest in Swing as a whole and lead you to delve more into some areas of it.

In this tutorial, through only a few quick lessons, we've managed to turn our example application, the flight reservation system, from a basic application that merely provided functionality to a more professional application that solved all the issues that Swing applications need to deal with. We started with an application that looked like this:



Departure	Destination	Flight Num	Number of Tix
AUS	DTW	4854	39
AUS	DTW	117	8
AUS	DTW	9315	17

And turned it into this:

Flight Reservation

Departure: AUS      Destination: IAH      Search

Departure	Destination	Flight Num	Number of Tixx
AUS	IAH	8906	35
AUS	IAH	0	11
AUS	IAH	3825	21

Number of Tickets:       Purchase

For completeness, let's review the important lessons learned from this tutorial:

- **Intermediate-level JTable:** In addition to learning how many of a table's properties can be changed to quickly change how a table looks, you also learned the following things about the JTable:
  - **TableRenderers:** You learned that you aren't constrained by the JTable's built-in color scheme for its cells. By creating your own `TableRenderer`, you can control how each cell appears in the table, when it is selected or deselected, and even how to provide feedback about the application (as when we grayed out unavailable flights).
  - **TableModel:** In addition to controlling the data, the `TableModel` can also control how data is aligned in the column, and whether a cell can be edited.
  - **TableEditors:** The JTable has many built-in editors for common classes that appear in data (`Strings`, `Integers`), but creating a custom editor for other data is difficult and beyond the scope of this tutorial.
  - **TableSorter:** Users have come to expect that they'll be able to sort their table's data in a meaningful way by clicking on the column headers. The JTable that is shipped with Swing does not support sorting at all, but Sun provides a `TableSorter` class (that isn't shipped with the JDK) that takes care of sorting for you.
- **Thread safety:** Thread safety is an important issue in Swing, as poor thread management can lead to a user getting locked out of the application or an ugly gray rectangle appearing instead of the application

## UI.

- `SwingUtilities.invokeLater()` should be used whenever a thread besides the event-dispatch thread needs to perform UI work.
- The `SwingWorker` class (another class not shipped in the JDK but published by Sun) should be used on the event-dispatch thread whenever a time-consuming action (like querying a database) must be performed.
- **Custom components:** While Swing provides nearly every component you could want, sometimes these components don't exactly work the way you want them too, or sometimes even the many components of Swing aren't enough to provide the widget your application needs. Modifying an existing component allows you to change the way a Swing widget works in your application. The best way to do this is to subclass the existing Swing widget and override the `paintComponent()` method to tailor the component to your needs. Creating a new custom component allows you to create original one-of-a-kind components not contained in Swing. The steps to follow to create a new component are to identify the components you'll need, lay them out properly, get them working with each other to get the new component working properly, and finally to wrap a public API around the new component to make it easy to work with.
- **Custom look and feel:** Creating a custom look and feel is a very difficult, complex, and time-consuming process, and probably deserves a tutorial all its own. However, there are easy ways to quickly change some behavior in your application that you learned in this tutorial:
  - Use the `UIManager` to override the properties used in an existing `LookAndFeel`, and supply your own colors, fonts, and borders to tailor the existing look and feel to your needs.
  - Use the new `Synth LookAndFeel`, an addition to Swing that allows you to create custom look and feels far easier and quicker than before.

## Resources

### Learn

- See this quick [how-to on creating custom table cell editors](#).
- Visit the [Sun tutorial on Swing](#), which is a good follow-up to this tutorial and will cover any components not covered here.
- Read the [Swing Javadoc](#) to see all the possible functions you can call on your Swing components.
- The [JavaDesktop Web page](#) offers the newest techniques in Swing.
- John Zukowski's [Magic with Merlin](#) series and [Taming Tiger](#) series cover Swing and related topics regularly.
- Michael Abernethy has penned several more advanced Swing-related articles, including "[Ease Swing development with the TableModel Free framework](#)" (developerWorks, October 2004), "[Go state-of-the-art with IFrame](#)" (developerWorks, March 2004), and "[Advanced Synth](#)" (developerWorks, February 2005).
- You'll find articles about every aspect of Java programming, including all the concepts covered in this tutorial, in the developerWorks [Java technology zone](#).
- [eclipse.org](#) is the official resource for the Eclipse development platform. Here you'll find downloads, articles, and discussion forums to help you use Eclipse like a pro.
- The developerWorks Open source zone has an entire section devoted to [Eclipse development](#).

### Get products and technologies

- Download the [swing2.jar](#) used in this tutorial. This also includes the `TableSorter` and `SwingWorker` classes that Sun doesn't ship with the JDK but includes in the Swing documentation.

### Discuss

- The Client-side Java programming [discussion forum](#) is another good place for assistance with Swing.
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

## About the author

Michael Abernethy

Michael Abernethy is an IBM employee who is currently the test team lead for the WebSphere System Management team, based in Austin, Texas. Prior to this assignment, he was a UI developer working in Swing at multiple customer locations.