Menadżerowie ułożenia

Menadżerowie ułożenia są obiektami odpowiedzialnymi za sposób ułożenia komponentów w kontenerach, a co za tym idzie, za sposób, w jaki wyświetlane są one na ekranie. Java dostarcza kilku menadżerów ułożenia: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, GridLayout, CardLayout, SpringLayout. Obiekty tych klas zapewniają zachowanie schematu wyświetlania komponentów nawet wtedy, gdy zmieniany jest rozmiar okienka aplikacji (np. przez przeciąganie myszką).

Używanie menadżerów ułożenia

Domyślnie każdy kontener ma związany ze sobą właściwy mu menadżer ułożenia:

JPanel	FlowLayout
content pane (JApplet, JDialog, JFrame)	BorderLayout

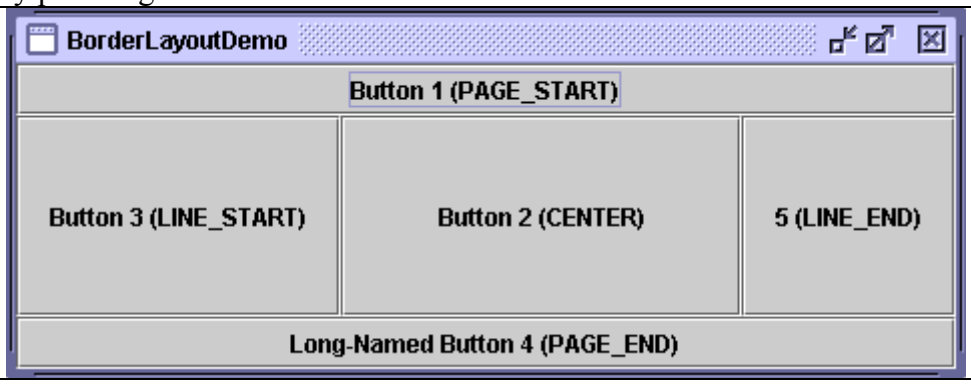
Aby zmienić menadżer ułożenia, należy wywołać metodę setLayout kontenera.

```
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());
```

Po ustawieniu menadżera ułożenia komponenty dodawane są do kontenera metodą add, której składnia i działanie zależy od typu ustawionego menadżera.

BorderLayout

Wygląd zapewniany przez tego menadżera ułożenia:



Domyślnie pomiędzy elementami w tym ułożeniu nie ma żadnego odstępu. Jednak można te odstępu zdefiniować, podając wartość odstępów w poziomie i pionie (wyrażoną w punktach):

1. w konstruktorze BorderLayout(int horizontalGap, int verticalGap),
2. albo w metodach void setHgap(int); void setVgap(int).

Sposób użycia (Java 1.5)

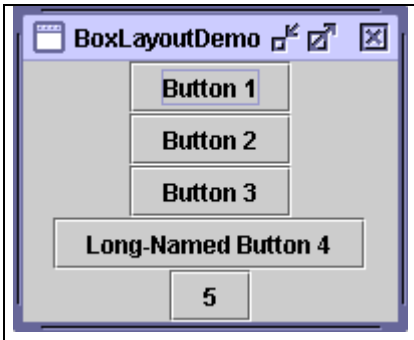
```
public static void addComponentsToPane(Container pane) {
    if (!(pane.getLayout() instanceof BorderLayout)) {
        pane.add(new JLabel("Container doesn't use BorderLayout!"));
        return;
    }
    if (RIGHT_TO_LEFT) {  // w klasie zadeklarowano: public static boolean RIGHT_TO_LEFT = false;
        pane.setComponentOrientation(
            java.awt.ComponentOrientation.RIGHT_TO_LEFT);
    }

    JButton button = new JButton("Button 1 (PAGE_START)");
    pane.add(button, BorderLayout.PAGE_START);

    // centralny komponent (duży rozmiar)
    button = new JButton("Button 2 (CENTER)");
    button.setPreferredSize(new Dimension(200, 100));
    pane.add(button, BorderLayout.CENTER);

    button = new JButton("Button 3 (LINE_START)");
    pane.add(button, BorderLayout.LINE_START);
```

```
        button = new JButton("Long-Named Button 4 (PAGE_END)");
        pane.add(button, BorderLayout.PAGE_END);

        button = new JButton("5 (LINE_END)");
        pane.add(button, BorderLayout.LINE_END);
    }
```

Wcześniejsze implementacje używały parametrów: north, south, east, west, center:

```
add(BorderLayout.CENTER, component)  // dopuszczalne, ale stare
add("Center", component)             //dopuszczalne, ale mogące kończyć się błędem
```

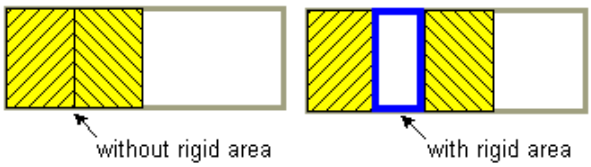
BoxLayout
Wygląd zapewniany przez tego menadżera ułożenia to: albo pojedyncza kolumna, albo pojedynczy wiersz komponentów. BoxLayout respektuje maksymalny żądany rozmiar komponentu oraz pozwala na wyrównywanie komponentów.

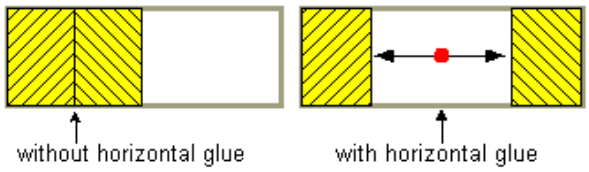



```
public static void addComponentsToPane(Container pane) {
    pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));

    addAButton("Button 1", pane);
    addAButton("Button 2", pane);
    addAButton("Button 3", pane);
    addAButton("Long-Named Button 4", pane);
    addAButton("5", pane);
}

private static void addAButton(String text, Container container) {
    JButton button = new JButton(text);
    button.setAlignmentX(Component.CENTER_ALIGNMENT);
    container.add(button);
}
```

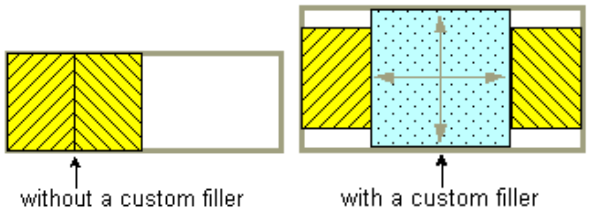
Niewidoczne komponenty jako wypełniacze
Odległość pomiędzy widocznymi komponentami umieszczonymi w kontenerze (z dowiązanym menadżerem ułożenia) można kontrolować deklarując odpowiednią obwiednię lub umieszczając w kontenerze niewidoczne komponenty. Klasą pomagającą tworzyć niewidocznych komponenty jest klasa Box. Wewnątrz tej klasy jest klasa zagnieżdżona, która dostarcza komponentów niewidoczych. Oto one:

Typ	Ograniczenia rozmiaru	Sposób tworzenia	

rigid area		Box.createRigidArea(size)	container.add(firstComponent);
			container.add(Box.createRigidArea(new Dimension(5,0)));
			container.add(secondComponent);
			 

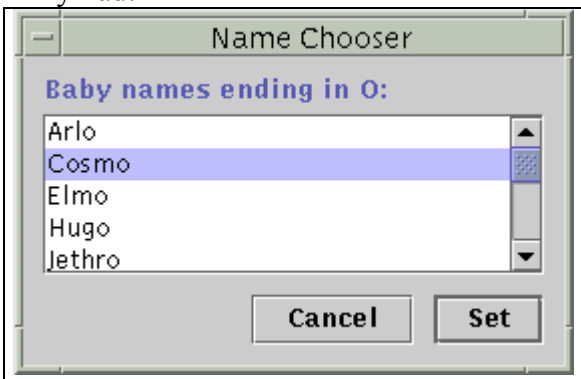
glue	poziomy		Box.createHorizontalGlue()	container.add(firstComponent);
				container.add(Box.createHorizontalGlue());
				container.add(secondComponent);
				 

	pionowy		Box.createVerticalGlue()	

Box.Filler użytkownika		deklarowany	new Box.Filler(minSize, prefSize, maxSize)	container.add(firstComponent);
				Dimension minSize = new Dimension(5, 100);
				Dimension prefSize = new Dimension(5, 100);
				Dimension maxSize = new Dimension(Short.MAX_VALUE, 100);
				container.add(new Box.Filler(minSize, prefSize, maxSize));
				container.add(secondComponent);
				 

RigidArea służy do tworzenia niewidocznych komponentów o zadanej wielkości.
Glue służy do tworzenia niewidocznych komponentów rozszerzalnych w kierunku poziomym lub pionowym (wypełniających całkowicie miejsce pomiędzy komponentami, z którymi sąsiaduje).
Box.Filler użytkownika jest niewidocznym komponentem, z zadeklarowanym (wg uznania) minimalnym, preferowanym i maksymalnym rozmiarem.

Przykład:




```
JScrollPane listScroller = new JScrollPane(list);
listScroller.setPreferredSize(new Dimension(250, 80));
listScroller.setAlignmentX(LEFT_ALIGNMENT);
...
//Lay out the label and scroll pane from top to bottom.
JPanel listPane = new JPanel();
listPane.setLayout(new BoxLayout(listPane, BoxLayout.PAGE_AXIS));
JLabel label = new JLabel(labelText);
...
listPane.add(label);
listPane.add(Box.createRigidArea(new Dimension(0,5)));
listPane.add(listScroller);
listPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

//Lay out the buttons from left to right.
JPanel buttonPane = new JPanel();
buttonPane.setLayout(new BoxLayout(buttonPane,
BoxLayout.LINE_AXIS));
buttonPane.setBorder(BorderFactory.createEmptyBorder(0, 10, 10,
10));
buttonPane.add(Box.createHorizontalGlue());
buttonPane.add(cancelButton);
buttonPane.add(Box.createRigidArea(new Dimension(10, 0)));
buttonPane.add(setButton);

//Put everything together, using the content pane's BorderLayout.
```

	Container contentPane = getContentPane();
	contentPane.add(listPane, BorderLayout.CENTER);
	contentPane.add(buttonPane, BorderLayout.PAGE_END);

FlowLayout

Ten menadżer ułożenia ustawia dodawane komponenty w linii, a jeśli trzeba, to tworzy nowy wiersz.
Konstruktory:

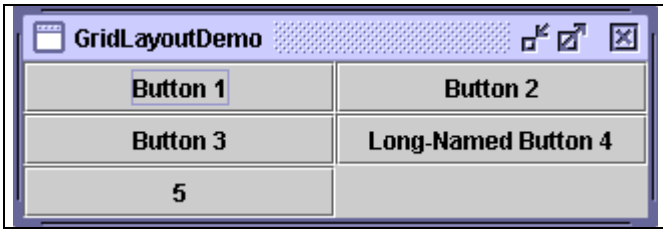
public FlowLayout(),
public FlowLayout(int alignment),
public FlowLayout(int alignment, int horizontalGap, int verticalGap)

gdzie:
alignment - parametr mówiący o sposobie wyrównania komponentów, który może przyjmować jedną z trzech wartości (FlowLayout.LEADING, FlowLayout.CENTER, FlowLayout.TRAILING);
horizontalGap, verticalGap - parametr mówiący o odstępach pomiędzy komponentami (domyślnie jest to 5 punktów).

	cp.setLayout(new FlowLayout());
	cp.add(new JButton("Button 1"));
	cp.add(new JButton("Button 2"));
	cp.add(new JButton("Button 3"));
	cp.add(new JButton("Long-Named Button 4"));
	cP.add(new JButton("5"));

GridLayout

Ten menadżer ustawia grupę komponentów o jednakowym rozmiarze w siatce o zadanej ilości wierszy i kolumn.

	pane.setLayout(new GridLayout(0,2));
	pane.add(new JButton("Button 1"));
	pane.add(new JButton("Button 2"));
	pane.add(new JButton("Button 3"));
	pane.add(new JButton("Long-Named Button 4"));
	pane.add(new JButton("5"));

GridBagLayout

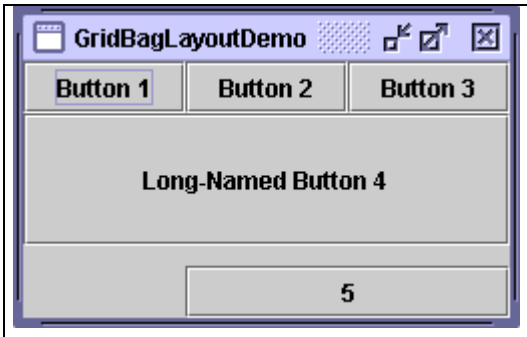
Jest to najbardziej rozbudowany i elastyczny menadżer ułożenia. Menadżer ten ustawia komponenty na siatce komórek, zezwalając, aby niektóre z nich rozciągały się na więcej niż jedna komórkę. Wiersze w siatce nie muszą mieć tej samej wysokości, podobnie jak kolumny nie muszą mieć tej samej szerokości. W ogólności: GridBagLayout umieszcza komponenty w prostokątach (komórkach) siatki, i później używa preferowanych rozmiarów komponentów do określenia, jak wielkie muszą być to prostokąty (komórki).

Ogólna orientacja siatki zależy od własności ComponentOrientation kontenera. Dla orientacji horyzontalnej, od lewej do prawej, współrzędna (0,0) siatki to lewy górny narożnik kontenera ze współrzędną x wzrastającą do prawej strony, oraz współrzędną y wzrastającą w dół siatki. Dla orientacji horyzontalnej, od prawej do lewej, współrzędna (0,0) siatki to prawy górny narożnik kontenera ze współrzędną x wzrastającą do lewej strony i współrzędną y rosnącą jak w poprzedniej orientacji.

Z każdym komponentem zarządzanym przez GridBagLayout związana jest instancja klasy GridBagConstraints. Instancja ta określa gdzie na siatce będzie pole wyświetlania dla danego komponentu, oraz jaka powinna być pozycja tego komponentu wewnątrz jego pola wyświetlania. Dodatkowo GridBagLayout bierze pod uwagę zdefiniowany w ograniczeniach minimalny oraz preferowane rozmiary komponentu.
Instancji klasy GridBagConstraints przekazuje się do menadżera ułożenia w metodzie add lub za pomocą metody setConstrains.

W przykładzie poniżej mamy siatkę trzech wierszy i trzech kolumn. Położenie i rozmiar komponentów określony jest przez podanie ograniczeń dla każdego z komponentów.

pane.setLayout(new GridBagLayout());
GridBagConstraints c = new
GridBagConstraints();
c.fill = GridBagConstraints.HORIZONTAL;
c.weightx = 0.5; c.gridx = 0; c.gridy = 0;
 	
button = new JButton("Button 1");
pane.add(button, c);
	GridBagLayout gridbag = new GridBagLayout();
	GridBagConstraints c = new
	GridBagConstraints();
	c.fill = GridBagConstraints.HORIZONTAL;
	
	pane.setLayout(gridbag);
	button = new JButton("Button 1");
	gridbag.setConstraints(button, c);
	pane.add(button);



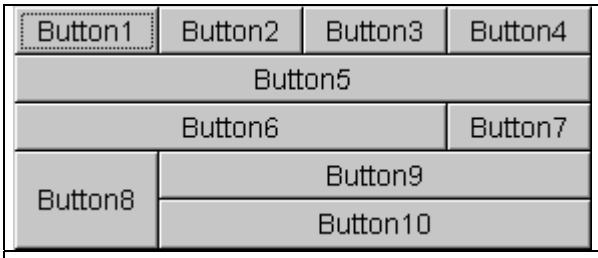

```
JButton button;
pane.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
c.fill = GridBagConstraints.HORIZONTAL;

button = new JButton("Button 1");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Button 2");
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Button 3");
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);

button = new JButton("Long-Named Button 4");
c.ipady = 40;      //utwórz wysoki komponent
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);

button = new JButton("5");
c.ipady = 0;       // powróć do ustawień domyślnych
c.weighty = 1.0;   //żadanie dodatkowej przestrzeni w pionie
c.anchor = GridBagConstraints.PAGE_END; //dół przestrzeni
c.insets = new Insets(10,0,0,0);  //top padding
c.gridx = 1;       // wyrównaj z button 2
c.gridwidth = 2;   // szeroki na 2 kolumny
c.gridy = 2;       // trzeci wiersz
pane.add(button, c);
```

Parametry GridBagConstraints:

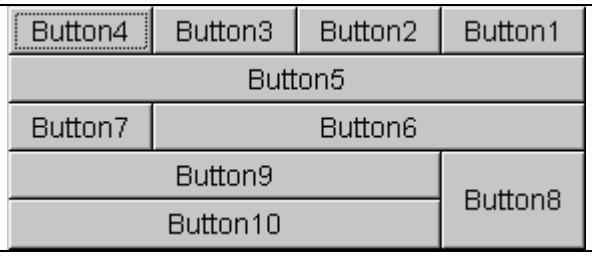
gridx, gridy	określa położenie wiodącego rogu wyświetlanego pola komponentu w układzie współrzędnych określonych przez orientacją kontenera. Położenie komponentu można również określać względem ostatnio dodanego komponentu. W takim przypadku, aby umieścić komponent dokładnie po prawej (dla gridx) lub dokładnie poniżej (dla gridy) ostatnio dodanego komponentu, używa się wartości GridBagConstraints.RELATIVE. Choć gridx oraz gridy przyjmują domyślnie tą wartość, zaleca

	się, aby jednak jawnie deklarować wartości tych parametrów.
gridwidth, gridheight	określa ilość kolumn (dla gridwidth) i wierszy (dla gridheight) obszaru rysowania komponentu. Inaczej mówiąc, ograniczenia te określają, ile komórek zajmuje komponent. Domyślną wartością jest 1. Do deklaracji komponentu, który ma być ostatnim komponentem w swoim wierszu (dla gridwidth) lub kolumnie (dla gridheight) używa się GridBagConstraints.REMAINDER. Do deklaracji komponentu, który ma być następnym po ostatnim w swoim wierszu (dla gridwidth) lub kolumnie (dla gridheight) używa się GridBagConstraints.RELATIVE.
fill	Used when the component's display area is larger than the component's requested size to determine whether (and how) to resize the component. Possible values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL (make the component wide enough to fill its display area horizontally, but don't change its height), GridBagConstraints.VERTICAL (make the component tall enough to fill its display area vertically, but don't change its width), and GridBagConstraints.BOTH (make the component fill its display area entirely).
ipadx, ipady	Specifies the component's internal padding within the layout, how much to add to the minimum size of the component. The width of the component will be at least its minimum width plus ipadx pixels. Similarly, the height of the component will be at least the minimum height plus ipady pixels.
insets	Specifies the external padding of the component -- the minimum amount of space between the component and the edges of its display area. The value is specified as an Insets object. By default, each component has no external padding.
anchor	Used when the component is smaller than its display area to determine where (within the display area) to place the component. There are two kinds of possible values: relative and absolute. Relative values are interpreted relative to the container's ComponentOrientation property while absolute values are not. Valid relative values are (since 1.4): CENTER (the default), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, and LAST_LINE_START. Valid relative values: NORTH, SOUTH, WEST, EAST, NORTHWEST, NORTHEAST, SOUTHWEST, SOUTHEAST . Here is a picture of how these values are interpreted in a container that has the default, left-to-right component orientation. --- |FIRST_LINE_START PAGE_START FIRST_LINE_END| | | | | |LINE_START CENTER LINE_END| | | | | |LAST_LINE_START PAGE_END LAST_LINE_END| ---
weightx, weighty	Used to determine how to distribute space, which is important for specifying resizing behavior. Unless you specify a weight for at least one component in a row (weightx) and column (weighty), all the components clump together in the center of their container. This is because when the weight is zero (the default), the GridBagLayout object puts any extra space between its grid of cells and the edges of the container. Generally weights are specified with 0.0 and 1.0 as the extremes: the numbers in between are used as necessary. Larger numbers indicate that the component's row or column should get more space. For each column, the weight is related to the highest weightx specified for a component within that column, with each multicolumn component's weight being split somehow between the columns the component is in. Similarly, each row's weight is related to the highest weighty specified for a component within that row. Extra space tends to go toward the rightmost column and bottom row.

Przykład z dziesięcioma komponentami zarządzanymi przez GridBagLayout w dwóch orientacjach:



Horizontal, Left-to-Right



Horizontal, Right-to-Left

Dla każdego z komponentów odpowiedni parametr fill, z instancji GridBagConstraints definiującej ograniczania, ma wartość GridBagConstraints.BOTH. Wartości innych parametrów, poza wartościami domyślnymi, wynosiły:

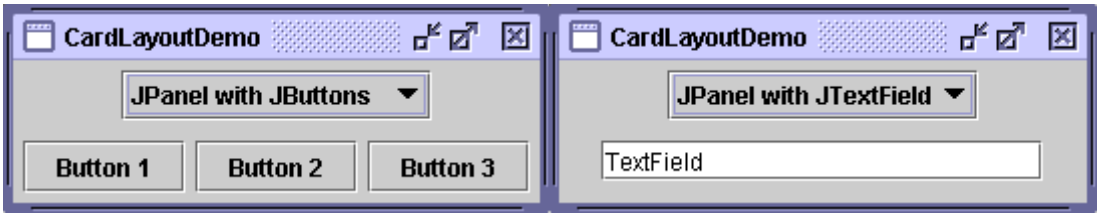
Button1, Button2, Button3: weightx = 1.0
Button4: weightx = 1.0, gridwidth = GridBagConstraints.REMAINDER
Button5: gridwidth = GridBagConstraints.REMAINDER
Button6: gridwidth = GridBagConstraints.RELATIVE
Button7: gridwidth = GridBagConstraints.REMAINDER
Button8: gridheight = 2, weighty = 1.0
Button9, Button 10: gridwidth = GridBagConstraints.REMAINDER

Kod źródłowy przykładu:

```
import java.awt.*;
 import java.util.*;
 import java.applet.Applet;

 public class GridBagEx1 extends Applet {

    protected void makebutton(String name,
                              GridBagLayout gridbag,
                              GridBagConstraints c) {
       Button button = new Button(name);
       gridbag.setConstraints(button, c);
       add(button);
    }

    public void init() {
       GridBagLayout gridbag = new GridBagLayout();
       GridBagConstraints c = new GridBagConstraints();

       setFont(new Font("SansSerif", Font.PLAIN, 14));
       setLayout(gridbag);

       c.fill = GridBagConstraints.BOTH;
       c.weightx = 1.0;
       makebutton("Button1", gridbag, c);
       makebutton("Button2", gridbag, c);
       makebutton("Button3", gridbag, c);

         c.gridwidth = GridBagConstraints.REMAINDER; //end row
       makebutton("Button4", gridbag, c);
```

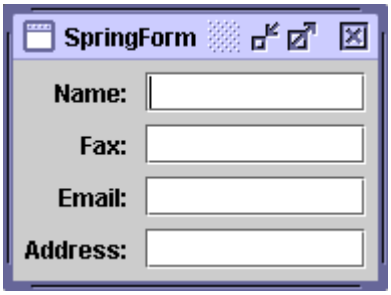
```
        c.weightx = 0.0;                //reset to the default
        makebutton("Button5", gridbag, c); //another row

        c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
        makebutton("Button6", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        makebutton("Button7", gridbag, c);

        c.gridwidth = 1;                //reset to the default
        c.gridheight = 2;
        c.weighty = 1.0;
        makebutton("Button8", gridbag, c);

        c.weighty = 0.0;                //reset to the default
        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        c.gridheight = 1;               //reset to the default
        makebutton("Button9", gridbag, c);
        makebutton("Button10", gridbag, c);

        setSize(300, 100);
    }

    public static void main(String args[]) {
        Frame f = new Frame("GridBag Layout Example");
        GridBagEx1 ex1 = new GridBagEx1();

        ex1.init();

        f.add("Center", ex1);
        f.pack();
        f.setSize(f.getPreferredSize());
        f.show();
    }
}
```

CardLayout



The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a tabbed pane, which provides similar functionality but with a pre-defined GUI.

SpringLayout



SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.

Threads and Swing

If your program creates and refers to its GUI the correct way, you might not need to worry about threads. If your program is an applet, it's safe to construct its GUI in the init method. You're also safe if your program is an application with the following common pattern:

```
//Thread-safe example
public class MyApplication {
    public static void main(String[] args) {
        JFrame f = new JFrame(...);
        ...//Add components to the frame here...
        f.pack();
        f.setVisible(true);
        //Don't do any more GUI work here.
    }

    ...
    //All manipulation of the GUI -- setText, getText, etc.
    //is performed in event handlers such as actionPerformed().
    ...
}
```

However, if your program creates threads to perform tasks that affect the GUI or if it manipulates an already visible GUI in response to anything but a standard event, read on.

The Single-Thread Rule

Swing components can be accessed by only one thread at a time, generally the event-dispatching thread. Thus, the single-thread rule is as follows.

Rule: Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread.

This rule might sound scary, but for many simple programs, you don't have to worry about threads.

Before we go into detail about how to write Swing code, let's define the term *realized*. "Realized" means that the component has been painted on-screen or that it is ready to be painted. A Swing component that's a top-level window is realized by having one of these methods invoked on it: setVisible(true), show, or pack. Once a window is realized, all the components it contains are realized. Another way to realize a component is to add it to a container that's already realized. You'll see examples of realizing components later.

Note: The show method does the same thing as setVisible(true).

Exceptions to the Rule

The rule that all code that might affect a realized Swing component must run in the event-dispatching thread has a few exceptions.

A few methods are thread safe.
In the Swing API documentation, thread-safe methods are marked with this text:

> This method is thread safe, although most Swing methods are not.

An application's GUI can often be constructed and shown in the main thread.
As long as no Swing or other components have been realized in the current runtime environment, it's fine to construct and show a GUI in the main thread of an application. To help you see why, here's an analysis of the thread safety of the thread-safe example. To refresh your memory, here are the important lines from the example:

```
public static void main(String[] args) {
    JFrame f = new JFrame(...);
    ...//Add components to the frame here...
    f.pack();
    f.setVisible(true);
    //Don't do any more GUI work here.
}
```

1. The example constructs the GUI in the main thread. In general, you can construct (but not show) a GUI in any thread, as long as you don't make any calls that refer to or affect already realized components.
2. The components in the GUI are realized by the pack call.
3. Immediately afterward, the components in the GUI are shown with the setVisible (or show) call. Technically the setVisible call is unsafe, because the components have already been realized by the pack call. However, because the program doesn't already have a visible GUI, it's exceedingly unlikely that a paint request will occur before setVisible returns.
4. The main thread executes no GUI code after the setVisible call. This means that all GUI work moves from the main thread to the event-dispatching thread, and the example is, in practice, thread safe.

An applet's GUI can be constructed and shown in the init method.
Existing browsers don't paint an applet until after its init and start methods have been called. Thus constructing the GUI in the applet's init method is safe, as long as you never call show() or setVisible(true) on the applet object.
These JComponent methods are safe to call from any thread: repaint and revalidate.
These methods queue requests to be executed on the event-dispatching thread.

Listener lists can be modified from any thread.
It's always safe to call the add*ListenerTypeListener* and remove*ListenerTypeListener* methods. The add/remove operations have no effect on an event dispatch that's under way.

System Properties

The System class maintains a set of properties, key/value pairs, that define traits or attributes of the current working environment. When the runtime system first starts up, the system properties are initialized to contain information about the runtime environment. including information about the current user, the current version of the Java runtime, and even the character used to separate components of a filename.

Here is a complete list of the system properties you get when the runtime system first starts up and what they mean:

Key	Meaning
"file.separator"	File separator (for example, "/")
"java.class.path"	Java classpath
"java.class.version"	Java class version number
"java.home"	Java installation directory
"java.vendor"	Java vendor-specific string
"java.vendor.url"	Java vendor URL
"java.version"	Java version number
"line.separator"	Line separator
"os.arch"	Operating system architecture
"os.name"	Operating system name
"os.version"	Operating system version
"path.separator"	Path separator (for example, ":")
"user.dir"	User's current working directory
"user.home"	User home directory
"user.name"	User account name

Your Java programs can read or write system properties through several methods in the System class. You can use a key to look up one property in the properties list, or you can get the whole set of properties all at once. You can also change the set of system properties completely.

Security consideration: Applets can access *some, but not all* system properties. For a complete list of the system properties that can and cannot be used by applets, refer to Getting System Properties□. Applets cannot write system properties.

System Properties that Applets Can Read

Applets can read the following system properties:

Key	Meaning
"file.separator"	File separator (for example, "/")
"java.class.version"	Java class version number
"java.vendor"	Java vendor-specific string
"java.vendor.url"	Java vendor URL
"java.version"	Java version number
"line.separator"	Line separator
"os.arch"	Operating system architecture
"os.name"	Operating system name
"path.separator"	Path separator (for example, ":")

To read a system property from within an applet, the applet uses the System class method getProperty. For example:

```
String newline = System.getProperty("line.separator");
```

The following applet reads all of the properties that are available to all applets:

You can find the source code (which is the same in both 1.0 and 1.1) in GetOpenProperties.java.

Forbidden System Properties

For security reasons, no existing browsers or applet viewers let applets read the following system properties.

Key	Meaning
"java.class.path"	Java classpath
"java.home"	Java installation directory
"user.dir"	User's current working directory
"user.home"	User home directory
"user.name"	User account name

Reading System Properties

The System class has two methods that you can use to read the system properties: getProperty and getProperties.

The System class has two different versions of getProperty. Both retrieve the value of the property named in the argument list. The simpler of the two getProperty methods takes a single argument: the key for the property you want to search for. For example, to get the value of path.separator, use the following statement:

```
System.getProperty("path.separator");
```

The getProperty method returns a string containing the value of the property. If the property does not exist, this version of getProperty returns null.

Which brings us to the next version of getProperty method. This version requires two String arguments: the first argument is the key to look up and the second argument is a default value to return if the key cannot be found or if it has no value. For example, this call to getProperty looks up the System property called subliminal.message. This is not a valid system property, so instead of returning null, this method returns the default value provided as a second argument: "Buy Java Now!"

```
System.getProperty("subliminal.message", "Buy Java Now!");
```

You should use this version of getProperty if you don't want to risk a NullPointerException, or if you really want to provide a default value for a property that doesn't have value or that cannot be found.

The last method provided by the System class to access properties values is the getProperties method, which returns a Properties□ object that contains the complete set of system property key/value pairs. You can use the various Properties class methods to query the Properties objects for specific values or to list the entire set of properties. For information about the Properties class, see Using Properties to Manage Program Attributes□.

Writing System Properties

You can modify the existing set of system properties using System's setProperties method. This method takes a Properties object that has been initialized to contain the key/value pairs for the properties that you want to set. Thus this method replaces the entire set of system properties with the new set represented by the Properties object.

The next example is a Java program that creates a Properties object and initializes it from this file: myProperties.txt.

```
subliminal.message=Buy Java Now!
```

The example program then uses System.setProperties to install the new Properties objects as the current set of system properties.

```
import java.io.FileInputStream;
import java.util.Properties;

public class PropertiesTest {
   public static void main(String[] args) throws Exception {
      // set up new properties object
         // from file "myProperties.txt"
      FileInputStream propFile = new FileInputStream(
                                    "myProperties.txt");
      Properties p = new Properties(System.getProperties());
      p.load(propFile);
```

```
            // set the system properties
            System.setProperties(p);
              // display new properties
            System.getProperties().list(System.out);
        }
    }
```

Note how the example program creates the Properties object, p, which is used as the argument to setProperties:

```
Properties p = new Properties(System.getProperties());
```

This statement initializes the new properties object, p with the current set of system properties, which in the case of this small program, is the set of properties initialized by the runtime system. Then the program loads additional properties into p from the file myProperties.txt and sets the system properties to p. This has the effect of adding the properties listed in myProperties.txt to the set of properties created by the runtime system at startup. Note that you can create p without any default Properties object like this:

```
Properties p = new Properties();
```

If you do this then your application won't have access to the system properties.

Also note that the value of system properties can be overwritten! For example, if myProperties.txt contains the following line, the java.vendor system property wil be overwritten:

```
java.vendor=Acme Software Company
```

In general, you should be careful not to overwrite system properties.

The setProperties method changes the set of system properties for the current running application. These changes are not persistent. That is, changing the system properties within an application will not affect future invocations of the Java interpreter for this or any other application. The runtime system re-initializes the system properties each time its starts up. If you want your changes to the system properties to be persistent , then your application must write the values to some file before exiting and read them in again upon startup.

Using Properties to Manage Program Attributes

An attribute has two parts: a name and a value. For example, "os.name" is the name for one of the Java platform's system attributes; its value contains the name of the current operating system, such as "Solaris".

The Properties□ class in the java.util package manages a set of *key/value pairs*. A key/value pair is like a dictionary entry: The key is the word, and the value is the definition. This is a perfect match for managing the names and values of attributes. Each Properties key contains the name of a system attribute, and its corresponding Properties value is the current value of that attribute. The System class uses a Properties object for managing system properties. Any Java program can use a Properties object to manage its program attributes. The Properties class itself provides methods for the following:

- Loading key/value pairs into a Properties object from a stream
- Retrieving a value from its key
- Listing the keys and their values
- Enumerating over the keys
- Saving the properties to a stream

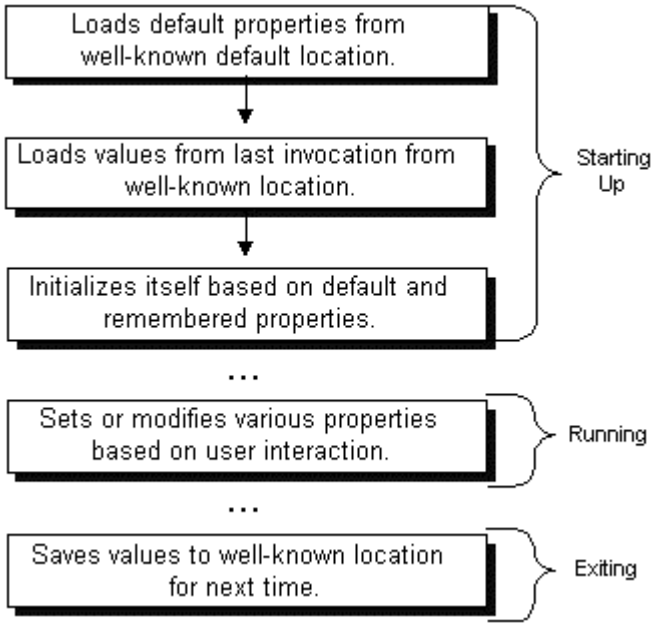
Properties extends the HashtableProperties□ class and inherits methods from it for doing the following:

- Testing to see if a particular key or value is in the Properties object
- Getting the current number of key/value pairs
- Removing a key and its value
- Adding a key/value pair to the Properties list
- Enumerating over the values or the keys
- Retrieving a value by its key
- Finding out if the Properties object is empty

Security Considerations: Access to properties is subject to approval by the current security manager. The example code segments in this section are assumed to be in standalone applications, which, by default, have no security manager. If you attempt to use this code in an applet, it may not work, depending on the browser or viewer in which it is running. See Security Restrictions□ for information about security restrictions on applets.

The Life Cycle of a Program's Properties

The following figure illustrates how a typical program might manage its attributes with a Properties object over the course of its execution.



Starting Up

The actions given in the first three boxes occur when the program is starting up. First, the program loads the default properties from a well-known location into a Properties object. Normally, the default properties are stored in a file on disk along with the .class and other resource files for the program.

Next, the program creates another Properties object and loads the properties that were saved from the last time the program was run. Many applications store properties on a per-user basis, so the properties loaded in this step are usually in a specific file in a particular directory maintained by this application in the user's home directory. Finally, the program uses the default and remembered properties to initialize itself.
The key here is consistency. The application must always load and save properties to the same location so that it can find them the next time it's executed.

Running

During the execution of the program, the user may change some settings, perhaps in a Preferences window, and the Properties object is updated to reflect these changes. For them to have a permanent effect, they must be saved.

Exiting

Upon exiting, the program saves the properties to its well-known location, to be loaded again when the program is next started up.

Setting Up Your Properties Object

The following Java code performs the first two steps described in the previous section: loading the default properties and loading the remembered properties:

```
. . .
// create and load default properties
Properties defaultProps = new Properties();
FileInputStream in = new FileInputStream("defaultProperties");
defaultProps.load(in);
in.close();

// create program properties with default
Properties applicationProps = new Properties(defaultProps);

// now load properties from last invocation
in = new FileInputStream("appProperties");
applicationProps.load(in);
in.close();
. . .
```

First, the application sets up a default Properties object. This object contains the set of properties to use if values are not explicitly set elsewhere. Then the load method reads the default values from a file on disk named defaultProperties.

Next, the application uses a different constructor to create a second Properties object, applicationProps, whose default values are contained in defaultProps. The defaults come into play when a property is being retrieved. If the property can't be found in applicationProps, then its default list is searched.
Finally, the code loads a set of properties into applicationProps from a file named appProperties. The properties in this file are those that were saved from the program the last time it was invoked (the next section shows you how this was done).

Saving Properties

The following example writes out the application properties from the previous example using Properties's store method. The default properties don't need to be saved each time because they never change.

```
FileOutputStream out = new FileOutputStream("appProperties");
applicationProps.store(out, "---No Comment---");
```

```
out.close();
```

The store method needs a stream to write to, as well as a string that it uses as a comment at the top of the output.

Note: The store method was introduced to the Properties class in JDK 1.2. If you are using an earlier release, use the save method instead.

Getting Property Information

Once you've set up your Properties object, you can query it for information about various keys/values that it contains. An application gets information from a Properties object after start up so that it can initialize itself based on choices made by the user. The Properties class has several methods for getting property information:

contains(Object *value*)
containsKey(Object *key*)
Returns true if the value or the key is in the Properties object. Properties inherits these methods from Hashtable. Thus they accept Object arguments. You should pass in Strings.
getProperty(String *key*)
getProperty(String *key*, String *default*)
Returns the value for the specified property. The second version allows you to provide a default value. If the key is not found, the default is returned.
list(PrintStream *s*)
list(PrintWriter *w*)
Writes all of the properties to the specified stream or writer. This is useful for debugging.
elements()
keys()
propertyNames()
Returns an Enumeration containing the keys or values (as indicated by the method name) contained in the Properties object.
size()
Returns the current number of key/value pairs.

Setting Properties

A user's interaction with a program during its execution may impact property settings. These changes should be reflected in the Properties object so that they are saved when the program exits (and calls the store method). You can use the following methods to change the properties in a Properties object:

put(Object *key*, Object *value*)
Puts the key/value pair in the Properties object.
remove(Object *key*)
Removes the key/value pair associated with key.

Both put and removeHashtable and thus take Objects. You should pass in Strings.