Chapter 20.AWTのコンポーネント

２０－１．コンポーネントクラス

２０－１－１．コンポーネントクラスの階層

AWTのGUIの部品としてアプレット上に配置され、何らかの操作の対象となる構造物は、コンポーネントと呼ばれていますが、本文の中でも今までButton,TextArea,あるいはTextFieldなどを紹介してきました。他にもいろいろなクラスが用意されていて、これらはすべてComponentというクラスのサブクラスとなっています。

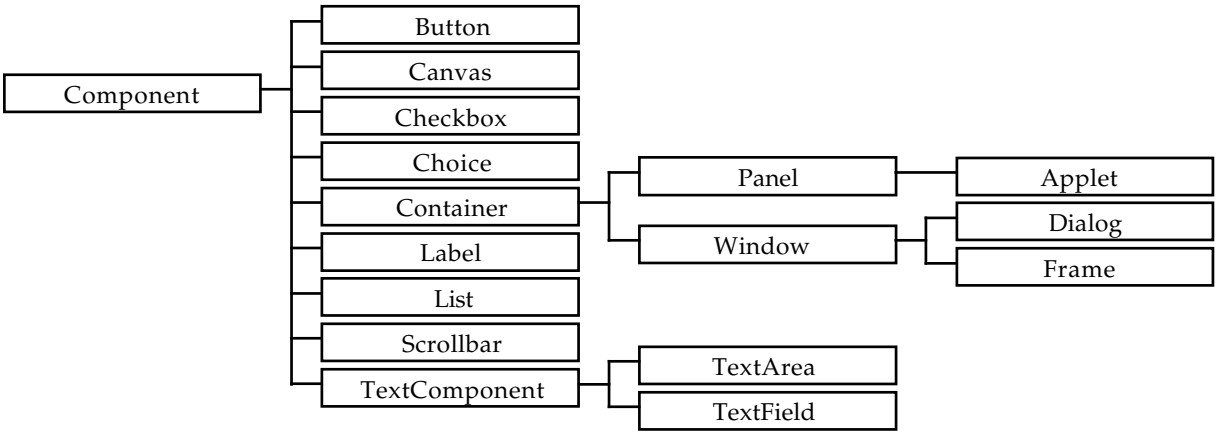

図 20-1 Component以下の主要なクラス（一部）

Componentクラスには、これらのGUIの部品に対しての共通して使われる特性がインスタンスメソッドとして用意されています。当然Componentクラスのメソッドは、それらのサブクラスのオブジェクトからも呼び出すことができます。上の図にあるように、アプレットもComponentクラスのサブクラスとなっています。

２０－１－２．Componentクラスの主要なメソッド

以下のメソッドは、上記のどのクラスでも用いることができます。

▼大きさについて
大きさや位置については、レイアウトマネージャによって、コンポーネントの位置や大きさは自動的に決まってしまいますが、レイアウトマネージャをOFFにすることにより、以下のようなメソッドを使って、明示的に場所や配置を行なうこともできます。以下の記述において、戻り値がない場合は型（**void**）を省略します。

Dimension getSize( )	大きさを返す
setSize(**int** *w*, **int** *h*)	大きさを設定する
setSize(Dimension *d*)	大きさを設定する

Dimensionクラスのオブジェクトのインスタンス変数height,widthを用いて、大きさを整数値で得ることができます。コンポーネントは配置されている必要があります。下の例は、ボタンを使っていますがが、他のコンポーネントでも同じようにメソッドを呼び出すことができます。

```
Button  b = new Button( "Hurry Up!" ); add( b );
Dimension  bsize = b.getSize( );
System.out.println( "Width: " + bsize.width + "  Height : " + bsize.height );
b.setSize( 100, 40 );
```

▼位置について

位置はコンポーネントを包含する外側のコンポーネント（コンテナと呼ばれています）上の座標となります。コンポーネントの左上の座標が対象となります。

Point getLocation( )	座標を得る
setLocation(**int** *x*, **int** *y*)	座標を設定する
setLocation(Point *p*)	座標を設定する

座標値はPointクラスのオブジェクトのインスタンス変数x, yによって知ることができます。

```
Point blocation = b.getLocation( );
System.out.println( "x axis: " + blocation.x + "  y axis : " + blocation.y );
b.setLocation( 10, 20 );
```

▼位置と座標を同時に

コンポーネントの左上の座標と幅・高さを同時に扱うメソッドです。今までの２つの機能を合わせたものになっています。Rectangleクラスには、それらの情報を示すための、x,y,width,heightというインスタンス変数が定義されています。

Rectangle getBounds( )	座標値・大きさの獲得する
setBounds(Rectangle *r*)	座標値・大きさを設定する
setBounds(**int** *x*, **int** *y*, **int** *width*, **int** *height*)	座標値・大きさを４つの値で設定

▼色関係のメソッド

色を指定しない限り、コンポーネントの色は標準の色（グレー）で設定されます。色指定をすることによって、ボタンの色や、テキストフィールドの文字の色などを変えることができます。

Color getBackground( )	背景色の獲得する
setBackground(Color *color*)	背景色を設定する
Color getForeground( )	描画色の獲得する
setForeground(Color *color*)	描画色を設定する

たとえば、アプレットでColor *backcolor* = getBackground( ); と記述するとアプレットの背景色を得ることができます。変数*b*で示されるボタンに対して*b*.setBackground(Color.yellow);と記述しますと、ボタン全体の色を変えることができます。あるいは、Color *forecolor* = getForeground( ); でアプレットの描画色をとれますし、*b*.setForeground(Color.red); と指定しますとボタンの文字色が赤色になります。

▼フォント関係のメソッド

フォントについては第８章で説明しました。ボタンやテキストフィールドの字体を変えることができます。

Font getFont( )	コンポーネントで現在用いられているフォントを返す
void setFont(Font *font*)	そのコンポーネントで用いるフォントを設定する

たとえば、変数*ta*で示されるテキストエリアに対して、*ta*.setFont(**new** Font("SansSerif", Font.BOLD, 18)); で入力される文字のフォントを変えることができます。

▼表示・利用制御のためのメソッド

パラメータの値が**true**の場合は、そのコンポーネントが見えていなければ、表示させたり、利用可能にします。通常は、配置された時点は表示・利用可能な状態になっています。**false**の場合は、そのコンポーネントを見せなくしたり、利用不可にします。

void setVisible(**boolean** *visibleSwitch*)	表示するかどうか設定する
boolean isVisible( )	見えているかどうかを得る
void setEnabled(**boolean** enableSwitch)	利用可能にするかどうかを設定する
boolean isEnabled( )	利用可能かどうか返す

そのコンポーネントの状態を得るメソッドでは、設定値と同様に**true**ならば、見えている・利用可能になっています。そのコンポーネントが利用可能であれば、イベントなどを発生させることができます。処理系によっては、利用可能でない場合は、表示の少し暗めに替えるなどの見た目上の措置も講じられます。

```
b.setVisible( true );          // ボタンbが表示されていなければ表示する
b.setEnable( false );          // ボタンbは見えているが、押せないようにする
```

▼描画領域や包含関係を求めるメソッド
この他に、これまでアプレットなどで使ってきたメソッドが用意されています。drawLineなどの描画などで必要なのが、次のようなメソッドです。

```
Graphics getGraphics( )   そのコンポーネント用の描画領域（グラフィックコンテキスト）を返す
Container getParent( )    そのコンポーネントが含まれている外側のコンポーネントを返します。
```

ただし、そのコンポーネントが配置され、画面上にないと**null**が返されます。以下の記述は、アプレットにおいて描画領域を得るものです。

```
Graphics g = this.getGraphics( );          // アプレットの描画領域を得る
g.drawLine( 10, 10 , 100, 100 );
```

▼描画関係
以下の２つのメソッドは、アプレットのときに出てきました。

```
paint( Graphics g )        描画用（オーバーライディングされる）
repaint( )                 再描画する
```

２０－１－３．いろいろなコンポーネントと、そのイベントリスナーの書き方

本文中で採り上げたボタン・テキストフィールド、テキスト領域以外にも、様々なコンポーネントが用意されています。それらのコンポーネントのうち、ユーザの操作（マウスクリック、テキスト入力）などを受け取り、それに対応したイベントを発生させるものがあります。そのようなコンポーネントは、発生したイベントを受け取り、対応するメソッドで処理するイベントリスナーを登録することができます（処理する必要がなければ登録しなくても構いません）。イベントリスナーに対応しているコンポーネントとしては、次のようなものがあります。

構造物	内容	利用するインターフェース
Button	ボタン	ActionListener
Checkbox	チェックボックス	ItemListener
Choice	選択メニュー	ItemListener
List	スクロールリスト	ActionListenerかItemListener
Scrollbar	スクロールバー	AdjustmentListener
TextArea	テキスト領域	TextListener
TextField	テキストフィールド	ActionListenerかTextListener

たとえば、アプレット上でChoiceを用意して、ユーザから何かを選んだときにイベントを発生させ、処理を行なうには、次のように、インターフェースをクラス定義に加え、対応する処理メソッドを用意させます。これは、第９章でボタンのときに、同様の処理を記述するやり方を既に学びました。また、各インターフェースに対応して、用意しなければならない処理メソッドの一覧を以下に挙げておきます。

インターフェース	実装するイベント処理用のメソッド
ActionListener	**public void** actionPerformed(ActionEvent *e*)
ItemListener	**public void** itemStateChanged(ItemEvent *e*)
TextListener	**public void** textValueChanged(TextEvent *e*)

AdjustmentListener　　**public void** adjustmentValueChanged(AdjustmentEvent *e*)

上記の例のようにインターフェースをクラス定義に加え、処理用のメソッドを登録すれば、コンポーネントに対してのユーザの操作を受け取ることができます。処理用のメソッドが登録されていないコンポーネントにユーザが操作を加えても何も起こりません（ただし、ボタンを押したりすることやテキストフィールドに文字を入力することなど、そのコンポーネントに固有の操作は行なえます）。

２０－２．各コンポーネントの詳細

以下に各コンポーネントについて紹介していきます。ただしCanvas、ScrollPaneやContainerなどを用いたスマートコンポーネントについては、別の章で紹介します。

２０－２－１．ボタン（Button）

以前の章でも紹介しました。レイアウトによって、表示形態は異なりますが、ユーザからのマウス入力を得るものです。

▼コンポーネントを作るために次のようなコンストラクタがあります。

Button( )	ラベルなしのボタンを作る
Button(String *label*)	指定されたラベルのボタンを作る

▼ボタンに表示されているラベル（表示文字列）についてのメソッド

String getLabel( )	ラベルを獲得する
setLabel(String *label*)	ラベルを設定する

▼イベントリスナーの追加・削除、およびアクションコマンドの獲得と設定のためのメソッド

addActionListner(ActionListener *listener*)	対処するリスナーを登録する
removeActionListner(ActionListner *listener*)	対処するリスナーを削除する
String getActionCommand( )	コマンドの文字列を設定する
setActionCommand(String *command*)	コマンドの文字列を獲得する

▼ボタンを使った例

信号機のように赤、緑、黄色のボタンを交互に表示するアプレットで、潜在的に３つのボタンがあり、押す度に表示されるボタンを変えるものです。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class SignalButtons extends Applet implements ActionListener {
        Button red, green, yellow;

        public void init( ) {
                green = new Button( "Green" );
                green.setBackground( Color.green );
                green.addActionListener( this );
                add( green );
                yellow = new Button( "Yellow" );
                yellow.setBackground( Color.yellow );
                yellow.addActionListener( this );
                add( yellow );
                red = new Button( "Red" );
                red.setBackground( Color.red );
                red.addActionListener( this );
                add( red );
                green.setVisible( false );
                yellow.setVisible( false );
        }
```

```
        public void actionPerformed( ActionEvent e ) {
            if ( red.isVisible( ) ) { red.setVisible( false ); green.setVisible( true ); }
            else if ( green.isVisible( ) ) { green.setVisible( false ); yellow.setVisible( true ); }
            else { yellow.setVisible( false ); red.setVisible( true ); }
        }
    }
```

２０－２－２．チェックボックス（Checkbox）

ボタンの一種ですが、チェックマークを付けるか付けないかというユーザの入力を受けるためのものです。

▼コンポーネントを作るために次のようなコンストラクタがあります。

Checkbox( )	空のラベルのチェックボックスを生成する
Checkbox(String *Label*)	指定されたラベルのチェックボックスを生成する
Checkbox(String *Label* , **boolean** *state*)	加えて、状態（onなら**true**、offなら**false**）も指定
Checkbox(String *Label* , **boolean** *state* , CheckboxGroup *group*)	加えて、指定されたチェックボックスグループに所属させる

▼表示や状態をみるためのメソッドとしては、次のようなものがあります。

String getLabel( )	チェックボックスのラベルを獲得する
setLabel(String *Label*)	チェックボックスのラベルを設定する
boolean getState( )	チェックの状態（on=**true**, off=**false**）を得る
setState(**boolean** *state*)	チェックの状態を設定する
CheckboxGroup getCheckboxGroup( )	属しているチェックボックスグループを得る
setCheckboxGroup(CheckboxGroup *group*)	チェックボックスグループに属させる

▼イベントリスナーを追加・削除するメソッドとしては、次のようなものがあります。

addItemListener(ItemListener *Listener*)	リスナーを登録する
removeItemListener(ItemListener *Listener*)	登録されているリスナーを削除する

▼チェックボックスを使った例

枠を描くボックス、内側を塗りつぶすボックス、色を反転させるボックスを用意して、四角形を描かせるようなアプレットを記述しました。ボックスに操作が加えられたときに起動されるitemStateChangedメソッドでは、ただ単にrepaintを呼び出すだけで、paintメソッドの方で各ボックスの状態をgetStateメソッドでチェックしています。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckTester extends Applet implements ItemListener {
    Checkbox frame = new Checkbox( "Frame", true ),
             fill = new Checkbox( "Fill" ),
             reverse= new Checkbox( "Reverse" );
    public void init( ) {
        frame.addItemListener( this );
        fill.addItemListener( this );
        reverse.addItemListener( this );
        add( frame ); add( fill ); add( reverse );
    }
    public void paint( Graphics g ) {
        super.paint( g );
        if ( fill.getState( ) ) { g.setColor( (reverse.getState( )) ? Color.blue : Color.red );
            g.fillRect( 20, 30, 100, 100 ); }
        if ( frame.getState( ) ) { g.setColor( (reverse.getState( )) ? Color.red : Color.blue );
            g.drawRect( 20, 30, 100, 100 ); }
    }
    public void itemStateChanged( ItemEvent e ) { repaint( ); }
}
```

２０－２－３．チェックボックスグループ（CheckboxGroup）

複数のチェックボックスを用意し、どれか１つのチェックボックスだけを選択させる（通称ラジオボタンと呼ばれている）ときに用いるクラスです。

▼コンポーネントを作るために次のようなコンストラクタがあります。

CheckboxGroup( )	チェックボックスグループを作る

▼表示や状態をみるためのメソッドとしては、次のようなものがあります。なお、getSelectedCheckboxでは、選ばれているチェックボックスがない場合は、**null**が替わりに返されます。

Checkbox　getSelectedCheckbox( )	選ばれているチェックボックスを返す
setSelectedCheckbox(Checkbox *box*)	指定されたチェックボックスを選ぶ

▼チェックボックスグループを使った例

以下の断片的な記述は、アプレットでチェックボックスグループを用意して、３つのフォント名を選ぶものです。実際にフォントを変えるには、ItemListenerを用いてitemStateChangedメソッドで変更させるようにします。

```
String    fontname [ ] =  {"Seif", "SansSerif", "MonoSpaced" };          // インスタン変数として
CheckboxGroup group= new CheckboxGroup( );
Checkbox    fontbox [ ] = new  Checkbox[  fontname.length ];

for  ( int i = 0;  i< fontname.length ; i++ ) {                          //  initメソッドの中で
        fontbox[ i ] = new Checkbox(  fontname[ i ] );
        fontbox[ i ].setCheckboxGroup(  group );  add( fontbox[ i ] ); }
```

２０－２－４．選択メニュー（Choice）

ポップダウン型のメニューで、ユーザに与えられた選択肢のうちから、一つを選ばせるために使います。

▼コンポーネントを作るために次のようなコンストラクタがあります。

Choice( )	空の選択メニューをつくる

▼表示や状態をみるためのメソッドとしては、次のようなものがあります。

String getItem(**int** *index*)	指定された番号の項目を返す
int getItemCount( )	メニューに項目がいくつあるか返す
int getSelectedIndex( )	現在選択されている項目の番号を返す
String getSelectedItem( )	現在選択されている項目の名前を返す
add(String *item*)	メニューに項目を追加する
insert(String *item* , **int** *index*)	指定された番号の後に項目を挿入する
remove(**int** *index*)	指定された番号の項目を削除する
remove(String *item*)	指定された名前の項目を削除する
removeAll( )	すべての項目を削除する
select(**int** *index*)	指定された番号の項目を選択する
select(String *item*)	指定された名前の項目を選択する

▼イベントリスナーを追加・削除するメソッドとしては、次のようなものがあります。

addItemListener(ItemListener *Listener*)	リスナーを登録する
removeItemListener(ItemListener *Listener*)	リスナーを削除する

▼選択メニューを使った例

５つの色から、色を選ばせる選択メニューを持ったアプレットの例で、描画される四角形の色がそれに応じて変化します。ItemListenerのitemStateChangedメソッドで変更しています。

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ColorSelector extends Applet implements ItemListener {
        Choice  colorselect;        int     selected = 0;
        String  names [ ] = {"red", "green", "blue", "orange", "yellow" }
        Color   colors [ ] = { Color.red, Color.green, Color.blue, Color.orange, Color.yellow };

        public void init( ) {
                colorselect = new  Choice( );
                for ( int  i=0; i< names.length ; i ++ ) {  colorselect.addItem( names[ i ] ); }
                colorselect.addItemListener( this );   colorselect.select( 0 );
                add( colorselect );
        }
        public void paint( Graphics g ) {
                super.paint( g );
                g.setColor( colors[ selected ] );
                g.fillRect( 20, 30, 100, 100 );
        }
        public void itemStateChanged( ItemEvent e ) {
                selected = colorselect.getSelectedIndex( );
                repaint( );
        }
}
```

２０－２－５．ラベル（Label）

ユーザにテキスト情報を見せるためにあります。ユーザがテキストを操作することはできません。ただし、文字列の変更などができるので、グラフィックコンテキストに直接描画するdrawStringよりはラベルを使った方がレイアウト上扱いやすいときもあります。なお、コンポーネントに共通のsetFontなどのメソッドを使ってフォントを個別に設定することもできます。

▼コンポーネントを作るために次のようなコンストラクタがあります。

Label( )	空のラベルを作る
Label(String *message*)	指定された文字列のラベルを作る
Label(String *message*, **int** *align*)	加えて、位置揃えをalignで指定する

▼ラベル記憶されている文字列や位置揃えのために次のようなメソッドがあります。

String getText( )	ラベルの文字列を獲得する
setText(String *message*)	ラベルの文字列を指定する
int getAlignment( )	ラベルの位置揃えを獲得する
setAlignment(**int** *align*)	ラベルの位置揃えを指定する

▼位置揃えのための整数値としては、次のようなものが（標準では左揃え）用意されています。

Label.LEFT…左揃え
Label.CENTER…中央揃え
Label.RIGHT…右揃え

▼ラベルを使った例
次の断面的な記述は、アプレットのinitメソッドで２つのラベルを追加するためのものです。１つは、コンストラクタですべてを指定していますし、もう１つは生成した後から指定しています。

```
add( new Label( "Please input name of  a fighter",  Label.CENTER ) );
Label  level = new Label( );
level.setText( 10 + " liters" );
level.setAlignment( Label.RIGHT );
add( level  );
```

２０－２－６．リスト（List）

ユーザに選択メニューと同じく与えられた選択肢の中から、選ばせるものですが、複数の項目を選択させることも可能です。使い方は、選択メニュー（Choice）と似ています。

▼コンポーネントを作るために次のようなコンストラクタがあります。引数の*multiple*が**true**のときは複数選択が可能になりです。**false**は複数選択不可となります。指定しなければ、複数選択不可になっています。

List( )	リストを作る
List(**int** *rows*)	行数を指定してリストを作る
List(**int** *rows*, **boolean** *multiple*)	複数選択可能なリストを作る

▼表示や状態をみるためのメソッドとしては、次のようなものがあります。複数の項目が指定されている場合は、配列として返されます。

String getItem(**int** *index*)	指定された番号の項目を返す
String [ ] getItems( )	リスト上のすべての項目を返す
int getItemCount( )	リストに項目がいくつあるか返す
int getRows( )	リストの行数を返す
int getSelectedIndex( )	選択されている項目の番号を返す
int [ ] getSelectedIndexes( )	選択されている複数の項目の番号を配列で返す
String getSelectedItem( )	選択されている項目の名前を返す
add(String *item*)	リストの最後に項目を追加する
insert(String *item* , **int** *index*)	指定された番号の後に項目を挿入する
remove(**int** *index*)	指定された番号の項目を削除する
remove(String *item*)	指定された名前の項目を削除する
removeAll( )	すべての項目を削除する
select(**int** *index*)	指定された番号の項目を選択する
select(String *item*)	指定された名前の項目を選択する
deselect(**int** *index*)	指定された番号の項目を選択を外す

▼イベントリスナーの追加・削除のメソッドとしては、次のようなものがあります。ActionListenerとItemListenerのどちらでも用いることができますが、ボタンなど他のコンポーネントがある場合は、アクションコマンドを設定することができませんので、ItemListenerを使った方が良いでしょう。

addItemListener(ItemListener *Listener*)	アイテムリスナーの登録
removeItemListener(ItemListener *Listener*)	アイテムリスナーの削除
addActionListner(ActionListener *Listener*)	アクションリスナーの登録
removeActionListner(ActionListner *Listener*)	アクションリスナーの削除

▼リストを使った例

次のアプレットはただ単に、ダブルクリックして選択された項目を文字端末に表示するものです。

```
import  java.applet.*;
import  java.awt.*;
import  java.awt.event.*;

public  class  ListTester  extends  Applet implements ActionListener {
        String  items [ ] = { "White", "Black", "Green", "Red", "Blue" };
        List  List = new  List( );
        public void init( ) {
                for ( int i = 0 ; i < items.length ; i++ ) {  List.add( items[ i ] ); }
                List.addActionListener( this ); add( List );
        }
        public void actionPerformed(  ActionEvent  e ) {
                System.out.println( List.getSelectedItem( ) );
        }
```

```
}
```

★スクロールバー（Scrollbar）
スクロールバーを表示し、タブの位置によってユーザに数量を決めさせることができます。スクロールバーを配置する場合は、後述するレイアウトではボーダーレイアウトにしておく必要があります。

▼コンポーネントを作るために次のようなコンストラクタがあります。

Scrollbar( )	スクロールバーを作る
Scrollbar(**int** *orientation*)	方向を指定してスクロールバーを作る
Scrollbar(**int** *orientation*, **int** *value*, **int** *visible*, **int** *minimum*, **int** *maximum*)	方向、値、タブの幅、最小値、最大値を指定して作る

▼表示や状態をみるためのメソッドとしては、次のようなものがあります。方向は、縦方向であればScrollbar.VERTICAL、横方向であればScrollbar.HORIZONTALという定数が用意されています。ブロック移動量（BlockIncrement）は、タブと矢印の間を空間を押したときの移動量になります。見た目のブロック移動量（VisibleAmount）は、表示されるタブの見た目の移動量（ドット数）になります。

int getBlockIncrement( )	ブロック移動量の獲得
int getMaximum( )	最大値の獲得
int getMinimum( )	最小値の獲得
int getOrientation( )	縦方向か横方向かを獲得
int getValue( )	値を獲得（これが一番良く使われる）
int getVisibleAmount( )	移動量の獲得
setBlockIncrement(**int** *block*)	ブロック移動量の設定
setMaximum(**int** *maximum*)	最大値の設定
setMinimum(**int** *minimum*)	最小値の設定
setOrientation(**int** *orientation*)	方向の設定
setValue(**int** *newValue*)	値の設定
setVisibleAmount(**int** *newAmount*)	移動量を設定する
setValues(**int** *value*, **int** *visible*, **int** *min*, **int** *max*)	４つの値を設定する

▼イベントリスナーの追加・削除のメソッドとしては、次のようなものがあります。

addAdjustmentListener(AdjustmentListener *Listener*)	リスナーの登録
removeAdjustmentListener(AdjustmentListener *Listener*)	リスナーの削除

▼スクロールバーを使った例
ラベルを１つ用意して、0〜100までの間で変わるスクロールバーのタブの位置に併せて、現在値を表示させるアプレットです。

```
import  java.awt.*;
import  java.awt.event.*;
import  java.applet.*;

public  class  ScrollbarTester  extends Applet implements AdjustmentListener {
        Scrollbar bar= new Scrollbar( Scrollbar.HORIZONTAL, 0, 10, 0, 100 );
        Label value = new Label( "" + bar.getValue( ) );
        public void init( ) {
                setLayout( new BorderLayout( ) );
                bar.addAdjustmentListener( this );
                add( "South", bar );
                 add( "Center", value );
        }
        public void adjustmentValueChanged(  AdjustmentEvent  e ) {
                value.setText( "" + bar.getValue( ) );
                repaint( );
        }
}
```

２０－２－７．テキスト領域（TextArea）

ユーザにテキストを編集させるために使います。テキストフィールドと共通のものは、こちらで扱います。使用例は第１７章を参照してください。

▼コンポーネントを作るために次のようなコンストラクタがあります。

TextArea( )	空のテキストエリアを作る
TextArea(String *text*)	文字列が設定されたテキストエリアを作る
TextArea(**int** *rows*, **int** *columns*)	行数・桁数を指定して
TextArea(String *text*, **int** *rows*, **int** *columns*, **int** *Scrollbars*)	すべてのパラメータを指定して

▼上記の最後のコンストラクタでのスクロールバーの指定は、次の４つが用意されています。基本的には、入りきらなくなると自動的にスクロールバーが表示されます。

TextArea.SCROLLBARS_NONE	なし
TextArea.SCROLLBARS_HORIZONTAL_ONLY	水平方向のみ
TextArea.SCROLLBARS_VERTICAL_ONLY	垂直方向のみ
TextArea.SCROLLBARS_BOTH	両方向

▼表示や状態をみるためのメソッドとしては、次のようなものがあり、以下のものはTextFieldと共通です。

String getText( )	テキストの獲得
setText(String *text*)	テキストの設定
int getCaretPosition( )	カーソルの位置の獲得（0～）
setCaretPosition(**int** *position*)	カーソルの位置の設定
String getSelectedText( )	選択された範囲のテキストを獲得
int getSelectedStart( )	選択された範囲の先頭の位置を獲得（0～）
int getSelectedEnd( )	選択された範囲の最後の位置を獲得（0～）
boolean isEditable( )	編集可能かどうか獲得（**true**なら編集可能）
setEditable(**boolean** *sw*)	編集可能かどうかの設定
select(**int** *start*, **int** *end*)	指定した範囲を選択
selectAll( )	すべてのテキストを選択
Dimension getMinimumSize( )	最小矩形サイズを得る
Dimension getPreferredSize( )	適性矩形サイズを得る

▼テキストエリア固有のメソッド

append(String *str*)	テキストの追加
insert(String *str*, **int** *pos*)	指定された位置の後にテキストを挿入
replaceRange(String *str*, **int** *start*, **int** *end*)	指定された範囲のテキストの置換
int getColumns( )	桁数を得る
setColoumns(**int** columns)	桁数を設定
int getRows( )	行数を得る
setRows(**int** rows)	行数を設定

▼イベントリスナーの追加・削除のメソッドとしては、次のようなものがあります。TextFieldと共通です。１文字変更される度にリスナーが呼び出されます。

addTextListener(TextListener *Listener*)	テキストリスナーの登録
removeTextListener(TextListener *Listener*)	テキストリスナーの削除

２０－２－８．テキストフィールド（TextField）

テキスト領域と同じくユーザにテキストを編集させるために使いますが、より簡略化され、一行だけ入力させるために用います。以下ではTextAreaと共通なものは省いています。例は第１７章を参照してください。

▼コンポーネントを作るために次のようなコンストラクタがあります。

TextField( )	空のテキストフィールドを作成
TextField(String *text*)	テキストを指定して作成
TextField(**int** *columns*)	桁数を指定して作成
TextField(String *text* , **int** *columns*)	テキストと桁数を指定して作成

▼表示や状態をみるための固有のメソッドとしては、次のようなものがあります。エコーキャラクタは、パスワードなどユーザの入力を隠したいときに使われます。

int getColumns( )	桁数の獲得
setColumns(**int** columns)	桁数の設定
char getEchoChar( )	エコーキャラクタの獲得
setEchoChar(char *echo*)	エコーキャラクタの設定
boolean echoCharIsSet( )	エコーキャラクタが設定されているか（**true**=設定）

▼テキストフィールド固有のイベント関係のメソッドとしては、次のようなものがあります。リターン（Return）キーあるいはEnterキーを押したときにActionEventを発生させることができるからです。

addActionListner(ActionListener *Listener*)	アクションリスナーの登録
removeActionListner(ActionListner *Listener*)	アクションリスナーの削除

２０－３．レイアウト

２０－３－１．基本的なレイアウト

アプレットやパネルなどのレイアウトを指定することができます。レイアウトはレイアウトマネージャーが行ないます。次のようなレイアウトが用意されています。レイアウトなしを指定しますと、各コンポーネントを座標で指定することができ、幅や高さも変えることができます。また、アプレットの標準では、フローレイアウトの中揃えが設定されています。レイアウトの設定によって、ボタンなどの表示がかなり変わりますので、注意してください。

```
▼レイアウトなし          setLayout( null );
▼フローレイアウト        setLayout( new FlowLayout( ) );                    // 中揃え
                         setLayout( new FlowLayout( FlowLayout.LEFT ) );    // 左揃え
                         setLayout( new FlowLayout( FlowLayout.RIGHT ) ); // 右揃え
▼ボーダーレイアウト      setLayout( new BorderLayout( ) );
```

ボーダーレイアウトでは、コンポーネントを配置するときに、以下のように上下左右（東西南北で記述します）、および中央のどの位置に置くか、指定することができます。

```
add( "North", new Button( "OK" ) );          // 上に配置
add( "South", new Button( "Back" ) );        // 下に配置
add( "West", new Button( "Left" ) ); // 左に配置
add( "East", new Button( "Right" ) ); // 右に配置
add( "Center", new Button( "Look" ) );       // 中央に配置
```

▼グリッドレイアウト

グリッドレイアウトでは、コンポーネントが付け加えられるたびに、上の行の左側から順次配置されていきます。addメソッドで付け加えられる順番に気をつけてください。

```
setLayout( new GridLayout( 2, 3 ) );                    //2行3列
setLayout( new GridLayout( 2, 3, 10, 15 ) );          //水平間隔10ドット、垂直間隔15ドット
```

２０－３－２．グリッドバッグレイアウト

アプレットのレイアウトをグリッドバッグレイアウトで指定することができる。これは、HTMLのTableタグと使い方は似ています。グリッドバッグレイアウトには、補助クラスとしてグリッドバッグ制約クラスが用いられ、一緒に使います。

```
GridBagLayout gb = new GridBagLayout( );
GridBagConstraints gc = new GridBagConstraints( );
```

ボタンなどを配置する場合は、このGridBagConstraintsクラスのオブジェクトをパラメータに取り、GridBagLayoutクラスのオブジェクトに用意されていますsetConstraintsメソッドを用いて、指定します。

```
Button button = new Button( "OK" );
gb.setConstraints( button, gc );
```

▼グリッドバッグ制約メソッドには、次のようなインスタンス変数（メンバー）が用意されています。

int	anchor;	配置の仕方
int	fill;	領域一杯に引き延ばすかどうかの指定
int	gridwidth, gridheight;	どれくらいのセルにまたがるか
int	gridx, gridy;	対象となる部品の行と列の指定
int	ipadx, ipady;	左右、上下に余裕を持たせるときの幅、高さ

・anchorが取り得る値としては、CENTER, EAST, NORTHEAST, NORTH, NORTHWEST, WEST, SOUTHWEST, SOUTH, SOUTHEASTがあります。GridBagConstraints.CENTER（中央に揃える）という感じで指定します。指定しなければCENTERになります。
・fillの値が取り得る値としては、BOTH, HORIZONTAL, VERTICAL, NONEがあります。指定しなければ、NONEになります。
・gridx, gridyは、0から始まることに注意しましょう。

▼グリッドバッグレイアウトを用いた例
たとえば、アプレット上につぎのようなレイアウトをしてみます。縦２行、横３列で、こんな感じです。initメソッドの中のレイアウトをする部分だけを記述します。

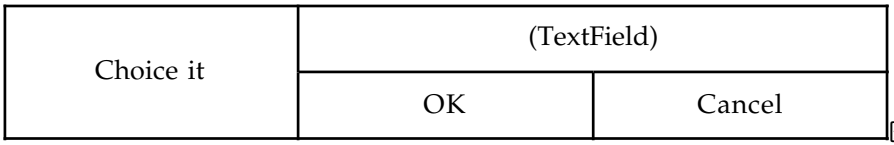

図22-2　例題レイアウト

```
public void init( ) {
    GridBagLayout gb = new GridBagLayout( );                    //レイアウトを作る
    GridBagConstraints gc = new GridBagConstraints( );          //制約を作る
    setLayout( gb );
    gc.fill = GridBagConstraints.BOTH;                          //縦横目一杯に引っ張り
    gc.ipadx = 10;      gc.ipady = 10;                          //縦横に余裕を10ドット
    gc.gridx = 0; gc.gridy = 0; gc.gridwidth = 1; gc.gridheight = 2;   //横1列縦２行にまたがる
    Button choice = new Button( "Choice It" );
    gb.setConstraints( choice, gc );                            // 制約の設定
    add( choice );                                              // Choice Itボタンの配置
    gc.gridx = 1; gc.gridy = 0; gc.gridwidth = 2; gc.gridheight = 1;   //横2列縦1行にまたがる
    TextField field = new TextField( 20 );
    gb.setConstraints( field, gc );                             // 制約の設定
    add( field );                                               // TextFieldの配置
    gc.gridx = 1; gc.gridy = 1; gc.gridwidth = 1; gc.gridheight = 1;   // OKボタンについて
    Button ok = new Button( "OK" );
    gb.setConstraints( ok, gc );
    add( ok );
    gc.gridx = 2; gc.gridy = 1;                                 // Cancelボタンについて
    Button cancel = new Button( "Cancel" );
    gb.setConstraints( cancel, gc );
    add( cancel );
}
```

２０－３．ウィンドウのためのクラス

Componentクラスの下に、他のコンポーネントを包含することができる以下のContainerクラス群が用意されています。これらを使ってウィンドウプログラミングを行なうことができます。Appletクラスも、この一員です。

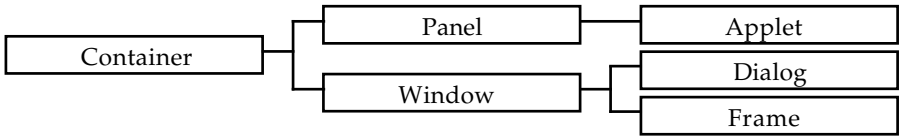

図20-2　Containerクラス以下の主要なクラス

２０－３－１．Panelクラス

★アプレットなどでサブパネルを利用する

アプレットは、１つのパネルになっていますが、見栄えをよくするためにサブパネルを生成することがあります。パネルは、アプレットの一部の領域を分割し、それ自体は通常のアプレットと同じように使うことができます。パネルの中では、レイアウトなどを変えることができるので便利です。以下にそのような記述の例を示しました。ただし、描画領域（グラフィックスコンテキスト）に直接描画するような場合は、スマートコンポーネントにする必要があります。

```
Panel     mypanel = new Panel( );                          //  パネルを生成
mypanel.setLayout(  new GridLayout( 3, 4, 10, 15 ) );      //  パネルの中だけグリッドレイアウトに
mypanel.add( new Button( "WOW" ) );                        //  パネルの中にボタンを配置
add(  mypanel  );                                          //  パネルをアプレット上に配置
```

▼Panelクラスを使った例
アプレットのinitメソッドで配置を行なう部分だけを記述してみます。アプレットの中は、グリッドレイアウトにして、パネルを３枚配置します。パネルの中は、フローレイアウトで、いろいろなコンポーネントを配置していきます。

```
public  void  init( ) {
        Panel     myPanel [ ]  = new Panel[  3 ];
        setLayout(  new GridLayout( myPanel.length, 1 ) );
        for ( int i=0;  i<myPanel.length ; i++) {
                myPanel[ i ] = new Panel( );
                add( myPanel[ i ] );
        }
        myPanel[ 0 ].add( new  Checkbox(  "Green" ) );
        myPanel[ 0 ].add( new  Checkbox(  "Red" ) );
        myPanel[ 0 ].add( new  Checkbox(  "Blue" ) );
        myPanel[ 1 ].add( new  TextField(  60 ) );
        myPanel[ 2 ].add( new Button( "OK" ) );
        myPanel[ 2 ].add( new Button( "Cancel" ) );
}
```

★カードレイアウト

さまざまなコンポーネントから構成されるアプレット上などで次々と画面を切り替えたい場合は、パネルクラスを併用して、それぞれをカードとして表示します。たとえば、次のように記述しますと２つのパネルを生成することができます。

```
CardLayout  card = new CardLayout( )
setLayout( card );
Panel     p1 = new Panel( );  add( "summer",  p1 );
Panel     p2 = new Panel( );  add( "winter",  p2 );
```

このパネルを切り替えるためには、showメソッドを利用します。

```
card.show( this, "summer" );                    // summerパネルの方を見せる
card.show( this, "winter" );                    // winterパネルの方を見せる
```

▼カードレイアウトを使った例
カードレイアウトを使ったサンプルのアプレットを記述してみました。それぞれのパネルには、ボタンだけしかありませんが、ボタンが押される度にパネルが切り替わっていきます。

```
import  java.awt.*;
import java.awt.event.*;
import  java.applet.* ;

public class SceneViwer extends Applet implements ActionListener {
        String    course [ ] = { "enterance", "room", "dining", "study", "bedroom", "exit" };
        Panel     scene [ ] = new Panel[ course.length ];
        CardLayout        scenario = new CardLayout( );
        int  cur = 0;

        public void init( ) {
                setLayout( scenario );
                for ( int i=0; i < course.length ; i++ ) {
                        Button  b = new  Button( course[ i ] );
                        b.addActionListener( this );
                        scene[ i ] = new  Panel( ); scene[ i ].add( Button  );
                        add( course[ i ], scene[ i ] );
                }
        }
        public void paint( Graphics g ) {
                scenario.show( this, course[ cur ] );
        }
        public void actionPerformed( ActionEvent   ae ) {
                cur = (cur+1) % course.length;
                repaint( );
        }
}
```

２０－３－２．Windowクラスのサブクラスとメニュー

ウィンドウ（Frame）は、第９章で取り上げました。スマートコンポーネントのものは、次の付録で説明します。なお、ウィンドウやダイアログは、標準設定ではボーダーレイアウトになっているので注意してください。ここでは、ダイアログとユーザインターフェースとして良く使われるメニューについて取り上げます。

★ダイアログ（Dialog）
ダイアログは、ウィンドウ（Frame）とほとんど同じですが、モーダルダイアログ（ModaDialog）にすることができるのが大きな違いです。モーダル指定をすると、入力が終了するまで（setVisibleでダイアログを隠すまで）、別のウィンドウに入力が移るのを禁止します。次のアプレットは、アプレット上のボタンが押されたら、ダイアログ上にボタンとテキストフィールドを出し、入力されたテキストを、アプレット上のテキストフィールドに入れるものです。ダイアログのコンストラクタの１番目のパラメータとして、ダミーのウィンドウを作っています。また、３番目の値が**true**になっているのがモーダルダイアログの指定です。

```
import  java.applet.*;
import  java.awt.*;
import  java.awt.event.*;

public class DialogTester extends Applet implements ActionListener {
        Dialog  dialog = new Dialog( new Frame( ),  "Enter Text", true);
        Button  change = new Button( "Change" ),  ok = new Button( "OK" );
        TextField  present = new TextField( "None", 20 ), input = new TextField( 30 );

        public void init( ) {
                change.setActionCommand( "Change" );
                change.addActionListener( this );
```

```
                ok.addActionListener( this );
                input.addActionListener( this );
                present.setEditable( false );
                add( present );
                add( change );
                dialog.setLayout( new FlowLayout( ) );
                dialog.add( input );
                dialog.add( ok );
        }
        public void actionPerformed( ActionEvent e ) {
                if ( e.getActionCommand( ).equals( "Change" ) ) { dialog.show( ); }
                else { dialog.setVisible( false ); present.setText( input.getText( ) ); repaint( ); }
        }
}
```

★メニュー（Menu）とメニューバー（MenuBar）

ウィンドウ（Frame）に対しては、メニューバーを（setMenuBarメソッドで）設定することができます。メニューバーには、複数のメニューを設定することができます。１つのメニューには、いくつかのメニュー項目（MenuItem）を登録しておきます。メニューの対処はActionListener（actionPerformedメソッド）で行ないます。メニュー項目には、アクションコマンドを登録しておけますので、それを見て、どのメニュー項目が選ばれたのか判断できます。メニューのメソッドは、Choiceクラスとだいたい同じになっています。次のアプレットは、メニューバー表示用のサブウィンドウを出して、メニューを選ばせます。選ばれたメニュー項目の文字列を、アプレット上のテキストフィールドに反映させます。

```
import  java.applet.*;
import  java.awt.*;
import  java.awt.event.*;

public  class  MenuBarTester  extends  Applet implements ActionListener {
        String  command [ ] = { "Start", "Stop", "Continue", "Reset" };
        MenuBar    mb = new   MenuBar( );
        Menu    menu = new   Menu( "File" );
        TextField tf = new TextField( 30 );
        Frame f = new Frame( "MenuBarTester" );

        public void init( ) {
                for ( int i = 0; i < command.length; i++ ) {
                        MenuItem  mitem = new MenuItem( command[ i ] );
                        mitem.addActionListener( this );
                        mitem.setActionCommand( command[ i ] );
                        menu.add( mitem );
                }
                mb.add( menu );
                f.setMenuBar( mb );
                f.setSize( 100, 50 );
                f.show( );
                add( tf );
        }
        public void actionPerformed( ActionEvent    e ) {
                tf.setText( e.getActionCommand( ) );
                repaint( );
        }
}
```