Chapter 15. キーボードとマウスのイベント処理

１５－１．キー入力やマウスの入力の扱い

１５－１－１．アプレットにおけるキー入力やマウスの入力

アプレットにおけるキー入力やマウス入力は、低レベルのイベント入力と呼ばれています。これらはボタンなどと違い、マウスやキーの状態に応じて、イベントが発生するのです。ボタンやメニューなどは、この低レベルのイベントを監視して、ボタンが押されたかどうかを見ています。

高レベルのイベント：　ボタン入力、テキスト入力、メニュー選択など
低レベルのイベント：　マウス入力、キー入力など

低レベルのイベント処理と、高レベルのイベント処理を１つのアプレットの中で行なうのは避けた方が無難かも知れません。アプレットの中で低レベルのイベント処理を行なってしまうと、高レベルのイベントが伝達されない可能性もあります。一方、共存することも可能です。たとえば、マウス入力とボタン入力を混在させた場合には、マウスがクリックされる位置がボタンの内部である場合には、ボタン入力のイベントが発生しますが、ボタン以外の場合にはマウス入力のイベントが発生します。

なお、ボタンなどのコンポーネントを用意した場合は、キー入力の対象がボタンに移ってしまいます。そのため、引き続きアプレットでキー入力を受け付けたい場合は、アプレット自身がキー入力を受け付けるように処理をしなければいけません。

１５－１－２．例レベルのイベント処理にインターフェースを利用する方法

これは、ボタンを使った方法と同じで、次のインターフェースを用います。このとき、以下の表に示されるようなメソッドを、アプレットのクラスの中に定義する必要があります。

KeyListener	キー入力に対処する
keyPressed(KeyEvent *e*)	キーが押されたときに呼び出される
keyReleased(KeyEvent *e*)	キーが離されたときに呼び出される
keyTyped(KeyEvent *e*)	キーが１回タイプされたときに呼び出される
MouseListener	マウス入力に対処する
mouseClicked(MouseEvent *e*)	マウスボタンがクリックされたときに呼び出される
mousePressed(MouseEvent *e*)	マウスボタンが押されたときに呼び出される
mouseReleased(MouseEvent *e*)	マウスボタンが離されたときに呼び出される
mouseEntered(MouseEvent *e*)	マウスポインタがアプレットの描画領域に入ったとき
mouseExited(MouseEvent *e*)	マウスポインタがアプレットの描画領域から出たとき
MouseMotionListener	マウスポインタの動きに対処する
mouseDragged(MouseEvent *e*)	マウスボタンがドラッグされたときに呼び出される
mouseMoved(MouseEvent *e*)	マウスポインタが動いたときに呼び出される

表15-1　低レベルのイベント用のインターフェース

インターフェースを使った場合は、このように必要のないメソッドまで定義しておかなければならないのですが、その分どのようなイベントが発生したときにどのメソッドが呼び出されるかの対応がわかりやすくなっています。

１５－２．キー入力への対応

１５－２－１．キー入力に対応するメソッドの定義

キー入力に対応するには、クラスの定義のところで、KeyListenerインターフェースを用います。次にアプレットがキー入力時に発生するイベントに対応することを登録する必要があります。この登録は、initメソッドの中でaddKeyListenerメソッドを使って行ないます。これは、ボタンのときに、addActionListenerを使って登録したのと同様です。ただし、今回は対象となるコンポーネントがアプレット自身なので、addKeyListenerメソッドの呼出しは対象を指定せず、以下のように直接呼び出します。

```
addKeyListener( this );          // アプレット上のキー入力をアプレット自身が対応する
```

このメソッドは、スーパークラスのAppletクラスで実装されていますので、repaint( )メソッドと同じようにオブジェクト名を省略して呼び出すことができます。以下のように、thisやsuper（この場合addKeyListenerがスーパークラスから継承されたメソッドなので）をつけて呼び出しても構いません。

```
this.addKeyListener( this );
```

KeyListenerインターフェースを利用する際に定義しなければならない次の３つのメソッドは、ユーザがキー入力が行なってイベントが発生したときに呼び出されるメソッドになっています。

▼キーイベントに対処するメソッドの書式

```
public void keyTyped( KeyEvent e ) {
        キーがタイプされたときに行なうこと
}
public void keyPressed( KeyEvent e ) {
        キーが押されたときに行なうこと
}
public void keyReleased( KeyEvent e ) {
        キーが離されたときに行なうこと
}
```

★キーがタイプされたときだけに限定する

マスクのときに一覧で挙げたように発生するキーイベントとしては、次の３つあります。この３つのイベントに対応して、それぞれのメソッドが呼ばれることになります。

キーが打たれた瞬間に発生するイベント	KEY_PRESSED →	keyPressed
キーが離された瞬間に発生するイベント	KEY_RELEASED →	keyReleased
キーが入力されたことを示すイベント	KEY_TYPED →	keyTyped

１つのキーが押されて、離されるまでに、これら３つのイベントが発生します。たとえば、小文字のmというキーをユーザが入力する際には、上の順番で３つのイベントが発生します。KEY_TYPEDは、離された後（KEY_RELEASEDの後）に発生することに注意してください。

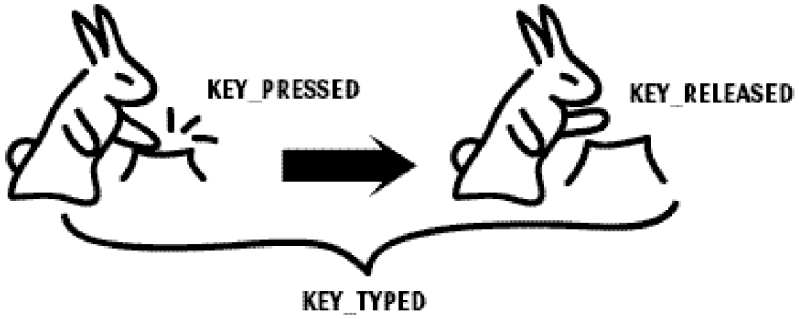

図15-1　キー入力と発生するイベント

KEY_TYPEDとKEY_PRESSEDの違いは、大文字のＡをキーボードで打つ場合には明確になります。大文字のＡを打つためには、SHIFTキーを予め押さなければなりません。SHIFTキーを押した段階でKEY_PRESSEDイベントが発生します。そして、小文字のaを押した段階でもKEY_PRESSEDイベントが続けて発生します。そして、aを離した段階でKEY_RELEASEDイベントが発生します。その後初めて、KEY_TYPEDイベントが発生し、そこでは大文字のAが押されたことが報告されます。

イベントの種類に対応するために、必要以外のメソッドは、何もしないように記述しておきます。たとえば、次の記述では、タイプされたときだけ何かを行ない、それ以外のとき（押された段階や離された段階で）は、何もしないように記述しています。

```
public void keyTyped( KeyEvent e ) {
                キーがタイプされたときに行ないたいこと
}
public void keyPressed( KeyEvent e ) { }          // 何もしない
public void keyReleased( KeyEvent e ) { }         // 何もしない
```

★通常の文字の場合

キーがタイプされたの場合は、キーイベントに対して、getKeyCharメソッドを呼び出してやれば、どのキーが押されたか文字として得ることができます。

```
char    c = e.getKeyChar( );
```

たとえば、ユーザのキー入力を得るためには、次のようにkeyTypedメソッドを記述します。これは、アプレット上で入力されたキーをシステムが用意した標準出力に表示しています（注１）。

```
public void keyTyped( KeyEvent e ) {
        char    c = e.getKeyChar( );
        System.out.println( "Input Character: " + c );
}
```

★特殊なキーの入力の場合

通常のキーの周りについている特殊なキーの場合には、文字としては得ることができません。イベントの種類がKEY_PRESSEDやKEY_RELEASEDの場合は、特殊なキーの入力を知ることができます。キーイベントに対して、getKeyCodeメソッドを呼び出します。このメソッドは、押されたキーに対応したシンボルコード（キーコード）を返してきます。キーコードは、整数型であることに注意してください。

```
int     keycode = e.getKeyCode( );
```

キーコードについては、KeyEventクラスに次のようなシンボルで定義されています。代表的なものをいくつかここで挙げましょう。すべてのキーに対応して定義されていますので、マニュアル（http://www.javasoft.com にあります）を参照してください。

KeyEvent.VK_HOME	Homeキー
KeyEvent.VK_END	Endキー
KeyEvent.VK_PAGE_UP	Page Upキー
KeyEvent.VK_PAGE_DOWN	Page Downキー
KeyEvent.VK_UP	↑キー
KeyEvent.VK_DOWN	↓キー
KeyEvent.VK_LEFT	←キー
KeyEvent.VK_RIGHT	→キー
KeyEvent.VK_CONTROL	Controlキー

KeyEvent.VK_SHIFT	Shiftキー
KeyEvent.VK_ALT	Alt（あるいはOption）キー
KeyEvent.VK_META	Metaキー

なお、Return（Enter）キーや、Backspaceキー、あるいはTABキーやEscapeキーなどは、Java言語で文字として定義してありますので、getKeyCharメソッドで文字として処理することも可能です。

たとえば、Homeキーが押されたのを判別して何か処理をさせるためには、次のようにif文を用います。

```
public void keyPressed( KeyEvent e ) {
        int keycode = e.getKeyCode( );
        if ( keycode == KeyEvent.VK_HOME ) {
                        // ホームキーが押されたときにすること
        }
}
```

１５－２－２．低レベルイベント処理におけるMVCプログラミング

低レベルのイベントの処理の場合でも、MVCプログラミングとしては同じようにアプレットのプログラムを設計します。インスタンス変数がモデルを示し、paintメソッドはビューを、そしてkeyTypedメソッドはコントロールを表しています。

・インスタンス変数にモデルとなる情報を格納しています
・paintメソッドでインスタンス変数の値を元に何かの表示を行ないます
・keyTyped（あるいはkeyPressed, keyReleased）メソッドでユーザの入力を得て、インスタンス変数の値を変えます

★押されたキーの文字を画面に表示する

通常の文字をただ単に画面に表示するためのアプレットを作ってみます。インスタンス変数 *c* がモデルになっています。paintメソッドでは、それを画面に表示しています。keyTypedでは、ユーザのキー入力をインスタンス変数に代入しています。

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

public class KeyIn extends Applet implements KeyListener {
        char    c;                                  // キー入力された文字を保持する

        public void init( ) {
                addKeyListener( this );     // キー入力にアプレット自身が対応する
                c = 'A';                    // 仮に最初はAとする。
        }

        public void paint( Graphics g ) {
                g.drawString( "Key In:  " + c, 100, 100 );
        }

        public void keyTyped( KeyEvent e ) {
                c = e.getKeyChar( );        // 打たれた文字を取り出す
                repaint( );                 // 再描画させる
        }

        public void keyPressed( KeyEvent e ) { }            // 押されたときは何もしない
        public void keyReleased( KeyEvent e ) { } // 離されたときも何もしない
}
```

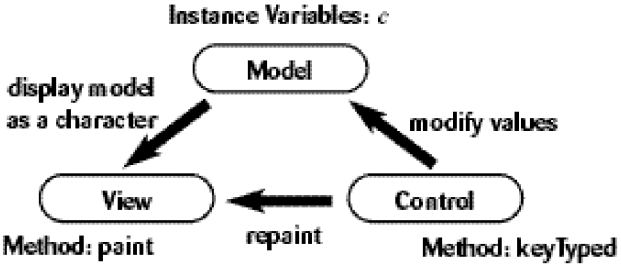

図15-2　キー入力アプレットにおけるMVC

（注１）　Webブラウザによっては、アプレットをアクティベート（顕在化）させないと、アプレットにキー入力がいかない場合があります。そのときは、マウスで一回アプレットが描画している領域のどこかをクリックしてください。あるいは、アプレットビューワなどでは、TABキーを押せばキー入力がアプレットに移ることがあります。

１５－３．マウス入力への対応

１５－３－１．マウス入力に対応するメソッドの定義

マウス入力に対応するには、クラスの定義のところで、MouseListenerインターフェースを用います。また、アプレットがマウスの入力に対応することを登録する必要があります。initメソッドの中でaddMouseListenerメソッドを呼び出して、登録を行ないます。このメソッドを呼び出すときも、コンポーネントはアプレット自身ですし、またアプレット自身がマウスの入力に対応しますので、次のように記述します。

```
addMouseListener( this ); // マウスの入力に対してアプレット自身が対応する
```

this.addMouseListener(**this**);　と記述してもいいのですが、アプレット自身に対してはthis.を省略できますので、上のように記述して構いません。このaddMouseListenerは、Componentクラスのメソッドなのですが、AppletクラスはComponentクラスを継承していますし、また皆さんが定義するアプレットは、Appletクラスを継承していますので、このメソッドを利用可能になっています。

MouseListenerに対応して、マウスイベントが発生したときに呼び出されるメソッドは次の５つのものを記述しなければなりません。

▼マウス入力に対応するメソッドの書式

```
public void mouseClicked( MouseEvent e ) {
        マウスがクリックされたときに行なうこと
}

public void mousePressed( MouseEvent e ) {
        マウスボタンが最初に押されたときに行なうこと
}

public void mouseReleased( MouseEvent e ) {
        マウスボタンが離されたときに行なうこと
}

public void mouseEntered( MouseEvent e ) {
        マウスが描画領域に入ったときに行なうこと
}
public void mouseExited( MouseEvent e ) {
        マウスが描画領域から出たときに行なうこと
}
```

★マウスイベントの種類を見分ける

キー入力と同じように、マウスに対して行なわれるさまざまな動作に対応して、イベントが発生します。たとえばマウスがクリックされたときだけ処理を行ないたい場合は次のmouseClickedメソッドを定義します。

```
public void mouseClicked( MouseEvent e ) {
        マウスがクリックされたときに行なうこと
}
```

あるいは、続けてドラッグ操作にはいるようなときは、ユーザはそのままボタンを押し続けることになります。クリック操作は、ボタンが押されて離されることを前提としています。そのような場合に、最初にマウスが押された場所を把握したければ、次のmousePressedメソッドを定義します。

```
public void mousePressed( MouseEvent e ) {
        マウスボタンが最初に押されたときに行なうこと
}
```

キー入力のときに発生するイベントと同様に、マウスボタンが押されて離される場合には、押されたときにMOUSE_PRESSEDイベントが発生し、離されるときMOUSE_RELEASEDイベントが発生し、その直後に一回クリックされたことを示すMOUSE_CLICKEDイベントが発生します。それぞれのイベントに対応してmousePressed、mouseReleased、およびmouseClickedメソッドが呼ばれることになります。

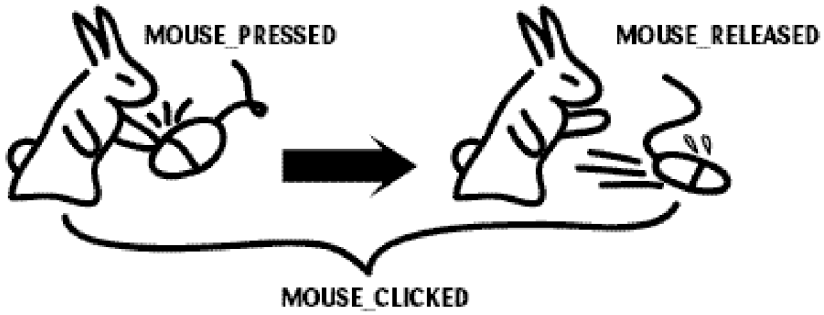

図15-3　マウス入力と発生するイベント

マウス入力の場合もキー入力と同様で対処する必要のないイベントに対しては、何もしないように記述します。たとえば、以下の記述は、マウスのクリックだけに対処するような記述の仕方になっています。

```
public void mouseClicked( MouseEvent e ) {
        マウスがクリックされたときに行なうこと
}
public void mousePressed( MouseEvent e ) { }          // 何もしない
public void mouseReleased( MouseEvent e ) { }         // 何もしない
public void mouseEntered( MouseEvent e ) { }          // 何もしない
public void mouseExited( MouseEvent e ) { }           // 何もしない
```

★マウスのイベントが起こった座標を得る

マウスのイベントが起こった座標を知るためには、受け取ったマウスイベントに対してgetXメソッド、getYメソッドを呼び出すことによって求めることができます。たとえば、変数eがマウスイベントを表している場合、次のように変数xとyに座標値を得ることができます。

```
int     x = e.getX( ), y = e.getY( );
```

たとえば、ある矩形（四角形）領域の中がクリックされたかどうかをチェックするためのプログラムの一部を記述してみましょう。インスタンス変数*checked*が論理値を保持する変数で、その値が**true**になっていれば、そ

の矩形の内部をユーザがクリックしたことを示すとします。このときに、mouseClickedメソッドは次のように記述します。記述では、矩形領域の左上の角が(10, 120)、右下の角が(40, 140)となっています。

```
public void mouseClicked( MouseEvent e ) {
        int x = e.getX( ), y = e.getY( );
        if ( x >= 10 && x <= 40 && y >= 120 && y <= 140 ) {
                checked = true ;
        } else {
                checked = false ;
        }
}
```

★マウスがクリックされた座標を画面に表示するアプレット

マウスがクリックされた場所のx座標、y座標の整数値を画面に表示するだけのアプレットを作成してみましょう。インスタンス変数の*currentx*と*currenty*に、クリックされた座標値を保持しておくものとします。ビューであるpaintメソッドでは、それを数値として表示するようにしています。

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

public class MouseIn extends Applet implements MouseListener {
        int currentx, currenty;          // x座標、y座標を保持

        public void init( ) {
                addMouseListener( this );          // マウス入力に対応する
                currentx=0; currenty=0;                    // 仮に最初は( 0, 0 )としておく
        }

        public void paint( Graphics g ) {
                g.drawString( "Mouse Clicked At x:" + currentx + " y:" + currenty , 50, 100 );
        }

        public void mouseClicked( MouseEvent e ) {
                currentx = e.getX( ); currenty = e.getY( );
                repaint( );
        }

        public void mousePressed( MouseEvent e ) { }          // 何もしない
        public void mouseReleased( MouseEvent e ) { }         // 何もしない
        public void mouseEntered( MouseEvent e ) { }          // 何もしない
        public void mouseExited( MouseEvent e ) { }           // 何もしない
}
```

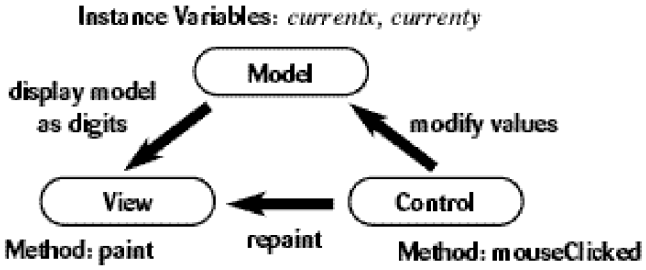

図15-4　マウス入力アプレットにおけるMVC

★キーにもマウスにも対応して四角形を表示している位置を変えるアプレット

今度は、クリックされた座標を数値で表示するのではなくて、そこに四角形を表示するようなアプレットを考えてみます。マウスだけではなくて、キー入力も受け付けましょう。矢印キーとマウスのクリックに反応し

て、四角形の表示位置を変えているようにしています。インスタンス変数の*currentx*と*currenty*が、それぞれ四角形を表示させるべき座標を保持しています。keyPressedおよびmouseClickedメソッドは共に、これらの値を変更して、repaintメソッドを呼び出して、再描画させています。また、インスタンス変数の*mycolor*とmouseEntered,mouseExitedの２つのメソッドを使って、マウスがアプレットの描画領域にあるときとないときで、色を変えています。

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

public class BoxMover extends Applet  implements  KeyListener, MouseListener {
        int     currentx, currenty;
        Color   mycolor;

        public void init( ) {
                addKeyListener( this );
                addMouseListener( this );
                currentx = 100;  currenty=100;  mycolor = Color.red;      // 最初は赤
        }

        public void paint( Graphics g ) {
                g.setColor( mycolor );
                g.fillRect( currentx, currenty, 50, 50 );
        }

        public void keyPressed( KeyEvent e ) {
                switch ( e.getKeyCode() ) {
                case KeyEvent.VK_UP:            currenty -= 5; break;
                case KeyEvent.VK_DOWN:          currenty += 5; break;
                case KeyEvent.VK_RIGHT:         currentx += 5; break;
                case KeyEvent.VK_LEFT:  currentx -= 5; break;
                default:  break;
                }
                repaint( );
        }

        public void mouseClicked(MouseEvent e)  {
                currentx = e.getX( ); currenty = e.getY( );
                repaint( );
        }
        public void mouseEntered(MouseEvent e)  {   mycolor = Color.red; repaint( ); }
        public void mouseExited(MouseEvent e)  {   mycolor = Color.blue; repaint( ); }

        public void keyTyped(  KeyEvent  e )  {  }
        public void keyReleased(  KeyEvent  e )  {  }
        public  void  mousePressed( MouseEvent   e ) { }
        public  void  mouseReleased( MouseEvent   e ) { }
}
```

１５－３－２．マウスが動くときに対応するメソッドの定義

マウスの動きに対応するには、クラスの定義のところで、MouseMotionListenerインターフェースを用います。アプレットがマウスの動きに対応することを登録するために、initメソッドの中でaddMouseMotionListenerメソッドを呼び出します。このメソッドの呼出しは、アプレット自身が対処することを示すために次のように記述します。

```
addMouseMotionListener( this );              // マウスの動きにアプレット自身が対応する
```

MouseMotionListenerインターフェースに対応して、マウスが動くイベントが発生したときに呼び出されるメソッドは次のように指定します。

▼マウスが動くときに対応するメソッドの書式

```
public void mouseDragged( MouseEvent e ) {
        マウスがドラッグされたときに行なうこと
}

public void mouseMoved( MouseEvent e ) {
        マウスが動いたときに行なうこと
}
```

２つメソッドがありますが、対処する必要のないイベント用のメソッドについては、何も記述する必要はありません。たとえば、次の記述はドラッグだけに対応しています。

```
public void mouseDragged( MouseEvent e ) {
        マウスがドラッグされたときに行なうこと
}

public void mouseMoved( MouseEvent e ) { }              // 何もしない
```

★マウスがドラッグされたところに線を表示する

今度は、マウスが押された場所からドラッグを行ない、その場所に線を表示するアプレットを作ってみましょう。ドラッグしている最中も、サイズが変化する線を逐一表示します。これは、ドローイングを行なうソフトウェアでは基本操作として用いられているものです。

まず、マウスボタンが押されたときに、mousePressedが呼び出されます。そこで、始点として、インスタンス変数*startx*、*starty*にその座標値を保持しておきます。しかし、クリックして終わりではなく、ボタンを押した後に、続けてドラッグ操作が始まり、線を描くことになります。この間、ユーザはマウスのボタンを離すわけではありませんので、イベントとしては、MOUSE_CLICKED（mouseClickedメソッドを呼ぶイベント）ではなくて、MOUSE_PRESSED（mousePressedメソッドを呼ぶイベント）の方を用います。

次に、ドラッグしている最中は、mouseDraggedが何回も呼び出されます。そこで、終点（現在のマウスの位置）としてインスタンス変数*endx*、*endy*にその座標値を保持しておきます。マウスを離した時点で、このメソッドは呼ばれなくなりますので、離す直前のマウスの座標値で最終的な線が描かれることになります。

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

public class LineDraw extends Applet  implements MouseListener, MouseMotionListener {
        int      startx, starty, endx, endy;                    // 始点と終点のx,y座標

        public void init( ) {
                addMouseListener( this );
                addMouseMotionListener( this );
                startx=0; starty=0; endx=0; endy=0;             // 0で初期化する
        }

        public void paint( Graphics g ) {
                g.drawLine( startx, starty, endx, endy );       // 線を描く
        }

        public void mousePressed( MouseEvent e ) {
                startx = e.getX( ); starty = e.getY( );         // 始点を設定
                endx = e.getX( ); endy = e.getY( );             // 終点も同じ座標に
                repaint( );
```

```
        }

        public void mouseDragged( MouseEvent e ) {
                endx = e.getX( ); endy = e.getY( );          // 終点だけを変更
                repaint( );
        }

        public void mouseClicked( MouseEvent   e ) { }
        public void mouseReleased( MouseEvent   e ) { }
        public void mouseEntered( MouseEvent   e ) { }
        public void mouseExited( MouseEvent   e ) { }
        public void mouseMoved( MouseEvent   e ) { }
}
```

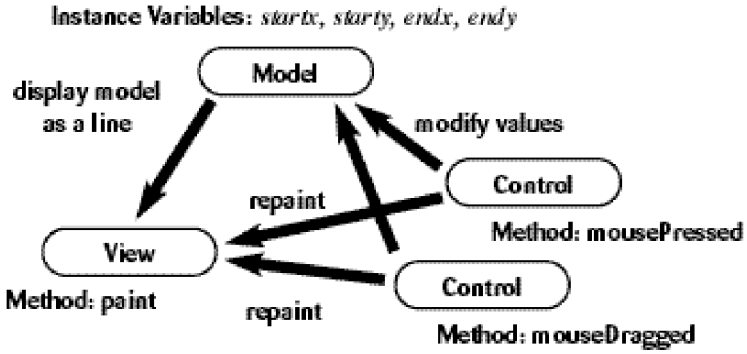

図15-5　線を表示するアプレットのおけるMVC

★マウスがドラッグされたところに四角形を表示する

線の替わりに四角形を描画してみましょう。四角形の場合は、paintメソッドを変更する必要がありますが、このときに問題がでてきます。四角形は、左上の座標を開始点として、右下方向に幅と高さを指定するdrawRectメソッドを使ってしか描けません。ですが、ドラッグは次の図のように、右下方向だけとは限りません。右上や、左上、あるいは左下といった方向にドラッグされる場合があります。そのため、４つの方向に対して、正確に四角形を描くために、**if～else if**文で場合分けをして描く必要が出てくるのです。各ドラッグの方向に関して、左上の角の座標をどの変数が保持しているのか場合わけしたのが、下の表です。これ以外に、幅と高さに関しても、考える必要があります。

ドラッグの方向	四角形の左上の角のx座標	四角形の左上の角のy座標
左上方向	*endx*	*endy*
左下方向	*endx*	*starty*
右上方向	*startx*	*endy*
右下方向	*startx*	*starty*

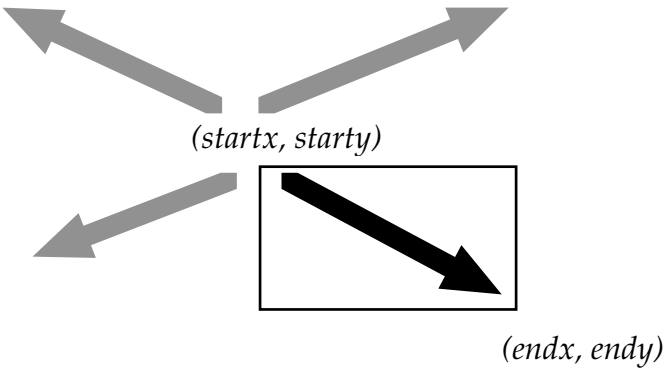

図15-6 マウスをドラッグする方向

四角形を描画するときのpaintメソッドは、次のようになります。その他の部分は、LineDrawアプレットとまったく同じです。

```
public void paint( Graphics g ) {
        if ( startx>endx && starty>endy ) {                    // 左上方向
                g.drawRect( endx, endy, startx - endx, starty - endy ); }
        else if ( startx>endx  ) {                             // 左下方向
                g.drawRect( endx, starty, startx - endx, endy - starty ); }
        else if ( starty>endy ) {                              // 右上方向
                g.drawRect( startx, endy, endx - startx, starty - endy ); }
        else {                                                 // 右下方向
                g.drawRect( startx, starty, endx - startx, endy - starty ); }
}
```

★配列で描いた線を覚えておく

マウス操作で線を描画するようなアプレットを作りましたが、一本線を引く度に前に引いた線が消えてしまいます。配列を使って、前に引いた線を10本までは覚えておくようにしましょう。startx、starty、endx、およびendyがそれぞれ始点と終点のx、y座標値を覚えておくための配列になっています。変数currentは、今まで線を引いた本数を表しているのですが、同時に新しい線の座標値を覚えるための配列のインデックスとしても使われています。paintメソッドでは、今まで引いた本数分、配列の要素に覚えた座標値を元に線を描いています。なお、10本まで線を引いたら、currentを0にして御破算にしています。

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

public class LineDrawer extends Applet implements MouseListener, MouseMotionListener {
        int     startx [] = new int[ 10 ], starty [] = new int[ 10 ];
        int     endx [] = new int[ 10 ], endy[] = new int[ 10 ];
        int     current = 0;

        public void init( ) {
                addMouseListener( this ) ;
                addMouseMotionListener( this );
        }

        public void paint( Graphics g ) {
                for ( int i=0; i<current; i++ ) {
                        g.drawLine( startx[ i ], starty[ i ], endx[ i ], endy[ i ] );
                }
        }

        public void mousePressed( MouseEvent  e ) {
                if ( current >= startx.length ) { current = 0; }
                startx[ current ] = e.getX( ); starty[ current ] = e.getY( );
                endx[ current ] = e.getX( ); endy[ current ] = e.getY( );
                current++;
                repaint();
        }

        public void mouseDragged( MouseEvent   e ) {
                endx[ current-1 ] = e.getX( ); endy[ current-1 ] = e.getY( );
                repaint();
        }

        public void mouseClicked( MouseEvent   e ) { }
        public void mouseReleased( MouseEvent   e ) { }
        public void mouseEntered( MouseEvent   e ) { }
        public void mouseExited( MouseEvent   e ) { }
```

```
    public void mouseMoved( MouseEvent e ) { }
}
```

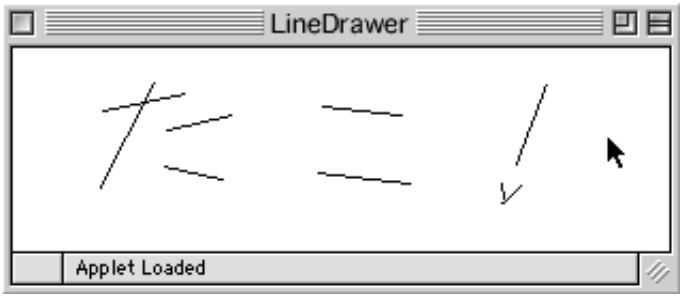

図15-7　10本まで線を描けるアプレットの実行例

１５－４．付属的な事項

１５－４－１．修飾キーのテスト

マウスがクリックされたときや、キーが入力されたときに、修飾キーの状態がどうなっているかをチェックすることができます。修飾キー（Modifier　Key）というのは、そのキー自体を押すことには意味がなく、別のキーを共に押されることによって、元のキーの意味を変えるという役割を持っています。たとえば、シフトキーは、英字と共に押されることによって、英字を小文字から大文字に変化させます。修飾キーとしては、つぎの４つが用意されています。

シフトキー	Shiftと書かれているキー
コントロールキー	Controlとか、Ctrlとか書かれているキー
Metaキー	Left/Rightキー、あるいはMacintoshのCommandキー
ALTキー	Windows用のキー、あるいはMacintoshのOptionキーなど

これらの修飾キーは、システムやアプリケーションによって、様々な意味や機能が割り振られています。アプレットの中で、ユーザからの入力イベントが発生したときに、これらの修飾キーの状態を確かめたい場合は、マウスイベントやキーイベントの共通のスーパークラスであるInputEventクラスに、これらの修飾キーの状態をチェックすることができるメソッドが用意されています。それを利用してみましょう。

isShiftDown( )	シフトキーが押されたかどうか
isControlDown( )	コントロールキーが押されたかどうか
isMetaDown( )	メタキーが押されたかどうか
isAltDown( )	ALTキーが押されていたかどうか

これらのメソッドは呼び出されると、指定された修飾キーが、押されていれば**true**を返します。押されていなければ、**false**を返します。そのため、これらのメソッドの呼び出しは、たいていはif文とともに用いられます。しかも、メソッドが論理値を返すので、論理値と比較する必要はありません。このように論理値を直接メソッド呼出し結果として返してくるものは、if文の条件式に直接と記述できるのできます。

```
if ( e.isShiftDown( ) == true ) {  .....      }        // こう書くこともできるが

if ( e.isShiftDown( ) ) {  .....      }             // 直接こう書くことができる
```

★マウスがクリックされたときに、どの修飾キーがおされていたかどうかチェックするアプレット

修飾キーが押されたのを確かめる例として、さきほどの、四角形を表示させるアプレットを前提として、そのプログラムの一部だけを修正するようにします。修飾キーが押されていたら、描く図形を変えるためには、描く図形を番号で覚えておくようなインスタンス変数を用意します。ここでは、shapeという名前の変数が宣言されているとします。

```
int     shape ;                    // 図形を整数値で覚えておくインスタンス変数
```

そこで、paintメソッドでは、この変数の値を参照しながら、描くものを変えるようにします。たとえば、shapeの値が３だったら、四角形の替わりに円を描くようにしています。paintメソッドでは、たとえば次のように記述できるでしょう（一部分だけを示しました）。

```
public void paint( Graphics g ) {
        if ( startx>endx && starty>endy ) {     // 左上方向
                if ( shape == 1 )     { g.drawRect( endx, endy, startx - endx, starty - endy ); }
                else if ( shape == 2 ) { g.fill3DRect( endx, endy, startx - endx, starty - endy, true ); }
                else if ( shape == 3 ) { g.drawOval( endx, endy, startx - endx, starty - endy ); }
                else { g.drawLine( startx, starty,  endx, endy ); }
        else if ( startx>endx  ) {
                ……
        }
}
```

一方、mousePressedメソッドでは、修飾キーを判定して*shape*に値を代入する部分を追加します。たとえば、コントロールキーがマウスをクリックする際に押していれば、*shape*の値を２に設定するようにしています。

```
public void mousePressed(  MouseEvent  e ) {
        currentx = e.getX( ); currenty = e.getY( );
        shape =  ( e.isShiftDown( ) ) ? 1 : ( e.isControlDown( ) ) ? 2 : ( e.isMetaDown( ) ) ? 3 : 4;
        repaint( );
}
```

★２つ以上のマウスボタンを持つコンピュータの場合

２つも３つも、マウスにボタンがある場合は、右ボタンや真ん中のボタンを押した場合は、Javaでは修飾キーを押しながら左ボタンを押したのであると認識されます。基本的には、どのマウスボタンが押されたのかチェックすることは避けた方がよいでしょう。なぜならば、一番マウスボタンが少ないシステムとしてMacintoshがあります。Macintoshではユーザを煩わせることがないように、マウスボタンが一つしかありません。あなたのアプレットは、Macintoshのユーザに実行されるかも知れません。彼のコンピュータには右ボタンも中ボタンもないのです。また、Webブラウザによっては、右ボタンを特別な機能のために用いているものもあります。結局、どのボタンが押されたかをチェックするよりも、上の修飾キーと組み合わせる方がより良い解決策になります。

なお、マウスイベントについては、クリックの場合は、何回クリックされたかということをgetClickCountメソッドを用いて確かめることができます。ダブルクリックや、トリプルクリックに対応することができます。たとえば、ダブルクリックのときに別の何かをさせたい場合は、次のように記述することになるでしょう。

```
public void mouseClicked(  MouseEvent  e ) {
        int count = e.getClickCount( );
        if ( count == 1 ) {             // シングルクリックのときにおこなうこと
                System.out.println( "Single Clicked" );
        }
        else if ( count == 2 ) {      // ダブルクリックのときにおこなうこと
                System.out.println( "Double Clicked" );
        }
}
```

１５－４－２．ボタンなどと組み合わせる

ボタンなどのコンポーネントと組み合わせる場合は、コンポーネントがキー入力を受け付けてしまいますので、ボタンを押してしまうと、以降キー入力が受け付けられなくなります。そこで、アプレット自身がキー入力を受け付けるために、requestFocus( )というメソッドを呼び出してやる必要があります。

以下のアプレットは、最初の文字を入力するアプレットですが、ボタンが押される度に、フォントサイズを大

きくしています。actionPerformedメソッドと、initメソッドでは、アプレット自身がキー入力を受け取るための指定をするために、requestFocusメソッドを呼び出しています。

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

public class KeyIn extends Applet implements KeyListener, ActionListener {
        Button  button;
        int     size = 10;
        char    c = 'A';

        public void init( ) {
                button =  new  JButton(  "Large" );
                button.addActionListener( this );
                add( button );
                addKeyListener( this );
                requestFocus( );
        }

        public void paint( Graphics g ) {
                g.setFont( new Font(  "Serif", Font.PLAIN,  size ) );
                g.drawString( "Key In:  " + c, 50, 50 );
        }


        public void actionPerformed(  ActionEvent  e ) {
                size = ( size - 10 + 2 ) % 100 + 10;       // 10ポイント〜110ポイントの間で、
                repaint( );                                // 2ポイントずつ大きくしていく
                requestFocus( );              // 再びアプレット自身がキー入力を貰う対象とする
        }

        public void keyTyped( KeyEvent  e ) {
                c = e.getKeyChar( );          // 打たれた文字を取り出す
                repaint( );                   // 再描画させる
        }

        public void keyPressed(  KeyEvent  e ) { }
        public void keyReleased(  KeyEvent  e ) { }

}
```

１５－５．課題

15-1
この講義で解説した例を実際に動作させてみたり、改良したりしなさい。たとえば、矢印キーで四角形を移動させるためにswitch文を使用した例を、一方の端まで四角形が表示されたら、反対の端から表示させるように改良しなさい。画面の端の幅と高さは、それぞれ200とします。

15-2
カラーテーブルの課題をさらに拡張して、各色の四角の部分をクリックしたら、対応するRGBの成分を表示するように改良しなさい。例えば、白色の格子のところをクリックしたら、アプレットのどこかに、

Red: 255 Green: 255 Blue: 255

と表示されるようにしなさい。

ヒント：
座標値を描かれている各四角形の間隔の大きさで割るなどして、赤成分と青成分の値を計算します。

15-3
四角形をドラッグして描く例題を拡張しなさい。四角形だけでなく、丸や直線などをドラッグして描けるようにしなさい。クラス名は、ShapeDrawerあたりで。

ヒント：修飾キーの例題を完成させてください。

15-4
マウスで何回かクリックし、クリックされた座標を頂点として、複合折れ線（Polyline）を描くようなアプレットを作りなさい。クラス名は、PolylineDrawerあたりで。

ヒント：配列で描いた線を覚えておくアプレットと同様に、座標を保持する配列を用意しなさい。

15-5
４つの点をクリックしたところにベジェ曲線を描くアプレットを作りなさい。クリックしたことがわかるように、クリックした場所には、小さい四角形を表示させるようにします。クラス名は、BezierDrawerあたりで。

15-6
上記のアプレットを修正・拡張して、Illustratorと同じようにドラッグしたら方向線を出すようにして描画していくように変更しなさい。複合曲線として、いくつもの曲線を描画できるようにします。また、Optionキーが押されたら、コーナーポイントにするようにしなさい。開始された頂点で再びクリック（およびドラッグ）されたら（頂点の近隣のある範囲を押されたら同じ頂点と判別します）、閉包された曲線形にします。一番最後の頂点の近隣で再びクリックされたら、そこで複合曲線を終了させます。再び、描画できるように、画面をクリアをするボタンも用意しなさい。

15-7
４つ以上の点をクリックしたところにβースプライン曲線を描くアプレットを作りなさい。クラス名は、SplineDrawerあたりで。