Chapter 19. メソッドの定義

１９－１．自作のメソッドを定義する

アプレットを定義する際に、initメソッド、paintメソッド、あるいはactionPerformedメソッドを定義してきました。これらのメソッドは、それぞれ役割を持っていました。アプレットが初期化されたとき、あるいは描画が要求されたとき、あるいはボタンが押されたときなど、特定のイベントに応じて呼び出されるという役割が決まっていました。

これらの役割が決められているメソッド以外に、クラスの中に自分で任意のメソッドを定義することができます。自分で新たに定義したメソッドは、上記のメソッドから更に呼び出されるという形で用いられます。たとえば、initメソッドの実行の中から、更に自作のメソッドを呼び出して使うという形で利用することができます。

そのような自作のメソッドが必要な場合はどのような場合なのでしょうか？たとえば、同じような作業を記述しなければならないのですが、少しずつ違うことをしたいときに、すべてをif文で場合分けして記述するよりも、自作のメソッドを定義した方がプログラムが短くなることがあります。自作のメソッドで、ある共通の部分だけを実現するようにしておき、それ以外の部分を呼出し側で分けて記述するのです。あるいは、大きなプログラムを制作するときに、仕事の分担をはっきりと分けて、自作のメソッドをいくつか定義して、各部分を担当するようにさせておくようにすれば、プログラムの構造がわかりやすいことがあります。

今までのプログラミング言語では、この章で述べるメソッドの役割を持っていた機構は、サブルーチン（Subroutine）あるいは関数（Function）と呼ばれていました。いずれにせよ、実際に自作のメソッドを定義する例をみていきましょう。メソッドを定義した方がわかりやすいと感じたら、自分で積極的に使っていってみましょう。失敗することもあるかも知れませんが、メソッドを使いこなせば、なぜプログラミング言語の発展の中でメソッドが重要な役割を持ってきたのか、それを身を持って経験することができることでしょう。

１９－１－１．メソッドの定義

メソッドの形は次の書式に従って定義します。これは、今までアプレットのinitメソッドやpaintメソッドで行なってきたものと同じです。

▼メソッド定義の書式

```
返す値の型　　メソッド名 ( 仮パラメータの宣言 ){
        実行する内容
}
```

メソッド名の後には、必ず丸括弧( )を記述します。メソッドはこの丸括弧があることによって、変数と区別されています。返す値がない場合は、**void**という型名が使われます。また、返す値の型の部分はクラス名でも構いません。仮パラメータの宣言は、メソッドが受け取るパラメータのことが記述されています。もしも、メソッドが受け取るパラメータがなければ、何も書かなくてよいのです。それでは、例として、今まで出てきてた予め役割を与えられたメソッドのいくつかを見てみましょう。

```
public void init( ) { }
```

これは、今まで記述してきたinitという名前のメソッドです。返す値の型が**void**になっています。ですから、このメソッドは何も値を返さないことを示しています。**void**の前に**public**がついています。これは、オブジェクトの外部からもこのメソッドを利用できるということを示していますが、詳しくは後の章で説明します。また、仮パラメータの宣言の部分には何も記述されていませんので、呼び出されたときには、何も受け取らないことを示しています。

```
public void paint( Graphics g ) {  }
```

何回も記述してきたpaintメソッドです。このメソッドも値を返さないので、**void**になっています。オブジェクト外部にも公開するので、**public**がついています。仮パラメータとして、*gc*という変数を宣言しています。この変数は、Graphicsというクラスのオブジェクトを指し示すように宣言されています。Graphicsクラスのオブジェクトは、アプレットの描画領域を表現しています。

★自作のメソッドを初めて作ってみる

それでは、メソッド定義の書式を使って自作のメソッドを作ってみましょう。呼び出されたら、文字端末にメッセージを出して少しの間休憩するというちょっとおさぼり気味のメソッドを作ってみましょう。休憩するために、Thread.sleepメソッドを用いましょう。このメソッドの定義は、paintメソッドなどと同様に、クラスの定義の中に記述します。

```
void shortSleep( ) {
        System.out.println( "Sleeping" );
        try {
                Thread.sleep( 500 );                    // 0.5秒休み
        } catch ( Exception e ) {
                System.err.println( e );
        }
        System.out.println( "Awaked" );
}
```

上記のメソッドは、文字端末にSleepingと表示してから、0.5秒休み、その後文字端末にAwakedと表示するだけの単純なメソッドです。何も返さないので、**void**と指定されています（外部に公開しませんので**public**はつけていません）。またパラメータを特に受け取りませんので、丸括弧の中には何も書かれていません。それでは、このメソッドをたとえばpaintメソッドから呼び出して使ってみましょう。

```
public void paint( Graphics g ) {
        g.setColor(  Color.red );
        g.fillRect( 10, 10, 100, 100 );
        shortSleep( );                          // 自作のメソッドを呼び出して、使ってみた
        g.setColor( Color.blue );
        g.fillRect( 10, 10, 100, 100 );
}
```

赤く塗りつぶされた四角形を表示してから、0.5秒休んで、青く塗りつぶされた四角形を表示するはずです。同時に、アプレットビューワが文字端末の出力用のウィンドウを用意していれば、その中にSleepingとAwakedが表示されています。呼出しは、メソッドの名前をそのまま書くだけで構いません。正式には、アプレットの中のメソッドを呼び出すということで、アプレットのオブジェクト自身を示す**this**を用いて、次のように記述することもできますが、大抵はこのようにきちんと書く必要はありません。

```
this.shortSleep( );                     // オブジェクト自身のメソッドを呼び出すことを明示する場合
```

★メソッドの実行について

先ほどの例について、メソッドの呼出しについて、どのような実行の順序になっているか図を使って追いかけていきましょう。まず、Webブラウザからアプレットのpaintメソッドが描画のために呼び出されます。実行の主体がpaintメソッドに移ります。paintメソッドでは、setColor、fillRectメソッドと順番に呼び出され、注目する自作のメソッドのshortSleepが次に呼び出されます。実行の主体は、shortSleepに移り、System.out.printlnやThread.sleepなどのメソッドが呼び出され、実行が進んで行きます。２番目のSystem.out.printlnを呼び出して、もうすることがなくなったら、呼出し側のpaintメソッドに戻ります。そして、次のsetColorが呼び出されていきます。メソッドは呼び出された場所の次の場所に戻るということを改めて確かめてください。

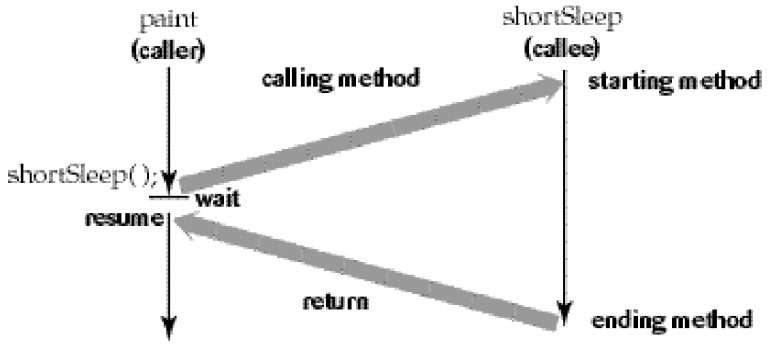

図19-1　shortSleepメソッドの呼出しと戻り

★メソッド名の衝突とシグネチャについて

さて、アプレットのメソッドとして使ってはいけない名前はどのような名前でしょうか？shortSleepは、既に役割を与えられているメソッドの名前と衝突することはありませんでした。ここで、アプレットでそのような役割を持っているメソッドの名前を挙げてみましょう。

> getAppletContext, getAppletInfo, getAudioClip, getCodeBase, getDocumentBase, getImage, getLocale, getParameter, getParameterInfo, init, isActive, play, resize, setStub, showStatus, start, stop, addNotify, add, addContainerListener, addImpl, addNotify, doLayout, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponents, getInsets, getLayout, getMaximumSize, getMinimumSize, getPreferredSize, invalidate, isAncestorOf, list, paint, paintComponents, paramString, print, printComponents, processContainer, processEvent, remove, removeAll, removeContainer, removeNotify, setLayout, update, validate, validateTree, ...

アプレットはAppletクラスのオブジェクトとして表現されています。このクラスは、PanelクラスやContainerクラスのサブクラスになっています。上ではContainerクラスまで辿ってみたのですが、一杯名前がありすぎて驚いたことでしょう。今までお馴染のメソッドも含まれています。この上にまだComponentクラスまでありまして、そこではさらにこの倍以上のメソッド定義されています。「既に特定の役割を持っているメソッドが一杯ありすぎるじゃないか！」とお嘆きの方もいるかも知れません。

ところが、Javaではメソッドの識別には、名前だけでなくて、仮パラメータの型なども見ています。たとえば、今までお馴染のpaintメソッドを自作のメソッドとして定義してみましょう。

```
void  paint(  int  x  ) {
        ........
}
```

これは、今まで定義してきたpaintメソッドとはちょっと違うようです。それは、パラメータの型が違うからです。Javaでは、同じ名前でも違うパラメータの型を持っているメソッドは別物とみなします。そのため、上のようなpaintメソッドを定義しても構わないのです。このように、メソッドを識別するために用いられるメソッドの特徴のことをシグネチャ（Signature）と呼んでいます。これは、次のような要素で構成されています。

▼シグネチャを決める要素
- ・メソッドの名前
- ・返す値の型
- ・仮パラメータの個数とそれぞれの仮パラメータの型

今までに、間違って**public void** paint( )などと記述して、コンパイラではエラーが出なかったけど、実行させると何も描画されなかったということはありませんでしたか？それは、シグネチャが違っていたのです。

シグネチャが違っていれば、メソッドは別のものだと認識されます。しかし、なるべく既存の名前と衝突しないように気をつけた方が無難でしょう。それはプログラムの読みやすさのためです。initやpaintといった名前

を別の自作のメソッドにつけた場合には、後で混乱することになる可能性は高くなります。

１９－１－２．メソッドのパラメータについて

メソッドに対して、描画するための領域や、座標などの情報を渡して、呼び出したい場合は多いでしょう。このときは、メソッドの名前にある丸括弧の中にパラメータを記述します。たとえば、簡単すぎて、あまり意味はないのですが、パラメータとして与えられた文字列にちょっとした前置きをつけて、文字端末に表示するメソッドを定義してみましょう。

```
void drawMessage( String message ) {
        System.out.println( "Received Message: " + message );
}
```

このメソッドでは、文字列をパラメータとして受け取り、それに仮に*message*という名前をつけています。これは、変数として使うことができます。このメソッドでは、この変数を更に別のメソッドSystem.out.printlnを呼び出すときのパラメータとして用いています。

アプレットのpaintメソッドから、このメソッドを利用してみましょう。アプレットから文字端末に出すという使い方は、実行テストの情報を出す以外にあまり行なわれませんし、アプレットをWebブラウザなどで実行した場合は、どこにも表示されないのですが、ここではアプレットをアプレットビューワなどで実行すると仮定して、使ってみます。実際のプログラム全体を記述してみましょう。drawMessageメソッドも、paintメソッドと同様に、クラスの定義の中に記述されているのがわかります。

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

public class MethodTester extends Applet {

        public void paint( Graphics g ) {
                drawMessage( "I am called from Applet Viewer." );
                drawMessage( "Do you understand on calling method?" );
        }

        void drawMessage( String message ) {
                System.out.println( "Received Message: " + message );
        }
}
```

文字端末へ出力できるようなアプレットビューワなどで、このアプレットを実行すると、描画要求がされる度に、paintメソッドからdrawMessageへ呼出しが行なわれ、次のように２行のメッセージが文字端末に表示されることになります。

```
Received Message: I am called from Applet Viewer.
Received Message: Do you understand on calling method?
```

★パラメータの評価とメソッド呼出し

パラメータを必要とするメソッドは、呼出し側で指定したパラメータを評価し、必ず最終的に定数に置き換えてから、呼び出されるメソッドに渡されます。すなわち、パラメータの評価を先に行なってから、メソッド呼び出しが行なわれるのです。第３章でも説明したように、一般的に、それぞれの側でのパラメータを区別するために、次のように呼び分けています。

実パラメータ（Actual Parameter）	呼出し側で指定した式や定数
仮パラメータ（Formal Parameter）	メソッド側で受け取るために指定した変数

パラメータは、引数（Argument）とも呼ばれることがありますので、仮パラメータのことは、仮引数（かりひきすう）と呼ぶことがあります。同じように実パラメータも、実引数（じつひきすう）と呼ばれることがあ

ります。

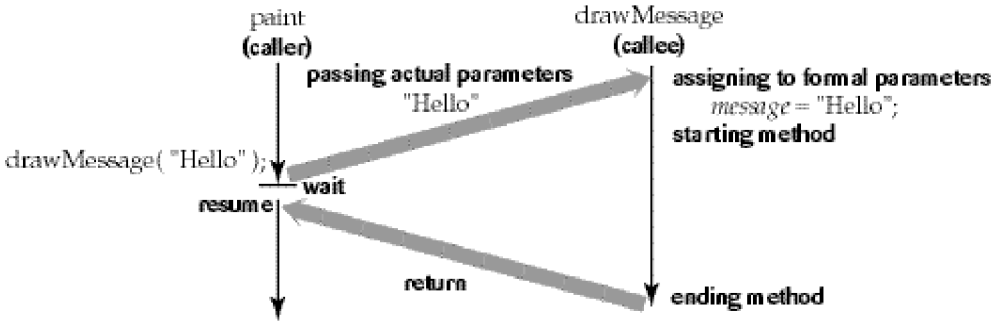

図19-2　drawMessageメソッドの呼出しにおける実パラメータと仮パラメータ

実パラメータが先に評価されてメソッドが呼び出される例を見てみましょう。実パラメータとして（計算）式を利用してみます。先ほどのdrawMessageを呼び出す際に、次のように記述されたとします。

```
int      bcount = 12;
drawMessage( "日本の戦艦は" + bcount + "隻である" );
```

この場合には、括弧内の実パラメータの式がまず評価されます。文字列の結合が行なわれ、評価結果として、"日本の戦艦は12隻である"という文字列定数が生成されます。呼び出されたdrawMessageメソッドの実行では、この文字列定数が仮パラメータの変数に代入されますので、最初に次のような代入が行なわれたの同じに状態で実行が開始されます。

```
message = "日本の戦艦は12隻である";
```

★パラメータを必要とするメソッドの使用例

drawMessageメソッドは、動作原理を説明するために用いましたが、もう少し実用的なメソッドを記述してみましょう。たとえば、二等辺三角形を書くようなメソッドを定義しておいて、これをpaintメソッドから呼び出せるようにしてみましょう。このメソッドに与えるべき情報は、次のようなものが考えられます。

・描画領域のオブジェクト
・頂点のx座標、y座標
・１辺の長さ

これだけの情報をパラメータとしてメソッドに渡せば、二等辺三角形（Δ：デルタとも呼ばれます）が描けます。次のように定義します。drawLineメソッドを使って３本の線を引いています。頂点から底辺までの距離は、底辺の長さと同一にしてあります。

```
void drawTriangle( Graphics g, int x, int y, int len ) {
        g.drawLine( x, y, x+len/2, y+len );          //右の斜辺
        g.drawLine( x+len/2, y+len, x-len/2, y+len ); // 底辺
        g.drawLine( x-len/2, y+len, x, y );                  // 左の斜辺
}
```

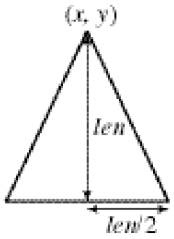

図19-3　drawTriangleメソッドの描く二等辺三角形

このメソッドをpaintメソッドから呼び出して使ってみましょう。まずは、定数値で１つだけ三角形描くように使ってみます。

```
public void paint( Graphics g ) {
        drawTriangle( g, 100, 100, 50 );                    // 定数値で
}
```

上記の呼出しでは、Webブラウザから変数*gc*という名前で受け取った描画領域をdrawTriangleの方にもそのまま渡しています。さらに、繰返しで実パラメータの値を変えながら、呼出しを行なってみましょう。どのような図が描かれるでしょうか？

```
void paint( Graphics g ) {
        for ( int i=10 ; i<50 ; i+=10 ) {
                drawTriangle( g, i, i, 50 );                    // 頂点の座標値を変えながら
        }
}
```

★更なる使用例

このようにパラメータを持ったメソッドを使えば、プログラムをかなりコンパクトにわかりやすくまとめられることができます。本当に、そうでしょうか？疑う人のために、前に出てきたプログラムをメソッドを使って書き直してみましょう。次のpaintメソッドは、以前の章で出てきたマウスに従って、四角形を描くアプレットのプログラムに含まれていました。

```
public void paint( Graphics g ) {
        if ( startx>endx && starty>endy ) { g.drawRect( endx, endy, startx - endx, starty - endy ); }
        else if ( startx>endx )         { g.drawRect( endx, starty, startx - endx, endy - starty ); }
        else if ( starty>endy )         { g.drawRect( startx, endy, endx - startx, starty - endy ); }
        else                            { g.drawRect( startx, starty, endx - startx, endy - starty ); }
}
```

これを変数shapeの値に従って、3D四角形や楕円、四角形を描くように拡張してみましょう。第11章では、3通り×4方向ということで、計12通りの場合分けを**if**文を使って行なわなければなりませんでした。思い出したでしょう？もしこの課題が宿題として出されてなければ、いけませんな！著者に担当の先生が誰だったかご連絡ください。さて、このようなとき、たとえば、drawShapeというメソッドを用意すれば、どうなるでしょうか？該当部分だけを書き直してみましょう。以下の記述で、変数*shape*は、インスタンス変数とします。

```
public void paint( Graphics g ) {
        if ( startx>endx && starty>endy ) { drawShape( g, endx, endy, startx - endx, starty - endy ); }
        else if ( startx>endx )         { drawShape( g, endx, starty, startx - endx, endy - starty ); }
        else if ( starty>endy )         { drawShape( g, startx, endy, endx - startx, starty - endy ); }
        else                            { drawShape( g, startx, starty, endx - startx, endy - starty ); }
}

void drawShape( Graphics g, int x, int y, int width, int height ) {
        if ( shape == 1 )               { g.drawRect( x, y, width, height ); }
        else if ( shape == 2 )          { g.draw3DRect( x, y, width, height, true ); }
        else                            { g.drawOval( x, y, width, height ); }
}
```

4方向＋3通りで両方のメソッドで併せて7通りの場合分けだけをするだけで済みました。えっ、5通り減っただけじゃないかっておっしゃりますか？じゃあ、たとえば４通り×５通り×６通りの組合せがある問題を想定してみてください。３つのメソッドで分担して場合分けをしたら、４＋５＋６で、計１５通りの場合分けをするだけで済みます。１つのメソッドでやったら、４×５×６＝１２０通りの場合分けをこつこつと書かなければいけないのです（注１）。このように場合分けを分担するだけで、組合せの掛け算を、足し算に置き換えることができるのです。これも、メソッドを使う１つの理由です。

★変数名の衝突

ローカル変数と仮パラメータとして受け取る変数を同じ名前にすると、ローカル変数の方が優先されてしまうので、仮パラメータの変数を利用することができなくなり、コンパイラがエラーを出します。

```
void dummy( int x , int y ) {
        int     x = 10;                         // 仮パラメータと同じ名前の変数を宣言した
        System.out.println( "result:" + x * y );
}
```

上の例では、パラメータの*x*とローカル変数の*x*が衝突（Collision）していますが、Java言語では、このような場合は許されず、コンパイラがエラーを出します。

★インスタンス変数との衝突

インスタンス変数と仮パラメータが同じ名前だったらどうなるでしょうか？次のアプレットのプログラムは、２つのメソッドを持っています。paintメソッドからdummyメソッドを呼び出しているのですが、dummyメソッドでは変数の*x*に10を代入しています。変数*x*としては、インスタンス変数（整数で最初に何も代入されていない場合は0が代入されています）とdummyメソッドの仮パラメータと両方に宣言されています。さて、System.out.printlnでは一体どのような値が文字端末に表示されるのでしょうか？

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

public class ShadowingTester extends Applet {
        int     x;

        public void paint (Graphics g ) {
                dummy( 20 );
                System.out.println( x );        // このxは、インスタンス変数の方を指す
        }

        void dummy( int x ) {
                x = 10;                         // このxは、仮パラメータ変数の方を指す
        }
}
```

インスタンス変数と同じ名前で仮パラメータの変数やローカル変数が宣言されている場合は、仮パラメータあるいはローカル変数の方が優先されることになります。この例では、dummyメソッドの中での代入は、仮パラメータの変数に行なわれています（注２）。インスタンス変数には何も代入されませんので、文字端末には、元々の値の0が表示されることになります。結局、仮パラメータ変数の*x*がインスタンス変数の方の*x*を見せなくしていました。これを変数の隠蔽（Shadowing）と読んでいます。

仮パラメータやローカル変数で同じ名前が使われていたときに、インスタンス変数の方を何とか使う方法はないでしょうか？思い出してください、アプレットそのものを指し示したい場合には、**this**という特別な変数が用意されていました。これを用いて、dummyメソッドの中で次のように記述すれば、たとえ同じ名前の変数が宣言されていたとしても、インスタンス変数の値を参照したり、変えることができます。

```
this.x = 10;                            // インスタンス変数の方のxに10を代入する
```

★多重に呼び出されるメソッド

自作のメソッドを複数定義した場合は、１つのメソッドから別の呼び出すことができます。たとえば、先ほどの三角形を描くメソッドを利用して、更に家を描くようなメソッドを定義してみましょう。二等辺三角形が家の屋根で、その下に家の壁と、窓を作ってみましょう。渡すパラメータとしては、描画領域と、家の屋根の頂点のx座標とy座標、家のサイズをの４つです。これらの情報をそのまま用いて、drawTriangleを呼び出してやります。そうすると、屋根の三角形が描かれますので、その下にdrawRectを用いて、壁と窓とを描画します。

家のサイズは、壁の１辺のサイズになっています。また、窓のサイズは壁のサイズの1/2としました。ちょうど、真ん中に窓を表示するために、x座標とy座標を微妙に調整しています。

```
void drawHouse( Graphics g, int x, int y, int size ) {
        drawTriangle( g, x, y, size );
        gc.drawRect( x - size / 2, y + size, size, size );
        gc.drawRect( x - size / 4, y + size + size / 4, size / 2, size /2 );
}
```

このメソッドをpaintメソッドから利用して、使ってみます。

```
public void paint( Graphics g ) {
        for ( int i=20 ; i<=170 ; i+=30 ) {
                drawHouse( g, i, 100, 20 );                    // 頂点のx座標値を変えながら
        }
}
```

結局、paintメソッドから、drawHouseメソッドが呼び出され、そこから更にdrawTriangleメソッドが呼び出されています。このように、プログラムは、自作のものあるいは既にJavaで定義されているメソッドの両方を含めて、メソッドの多重の呼出しの連鎖で処理が進んでいくことになります。これは、第３章で紹介しました。

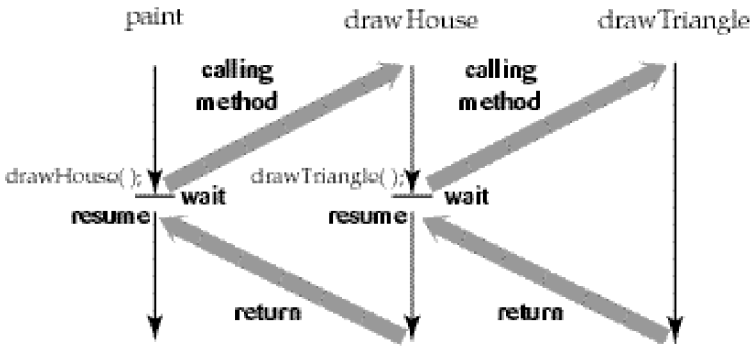

図19-4　多重に呼び出されるメソッド

※注１　それでも、力業でやる人は多いのです。よく1000通りの場合分けを１つのメソッドで行なって、「2000行ものプログラムを作った！」と愚かにも自慢する学生がいます。そのようなプログラムは、配列を使ったり、場合分けをメソッドで分担して書き直してみると、100行にも満たなくなります。
※注２　仮パラメータの変数にも値を代入できてしまいます。この場合は、元々保持していた値、すなわち、呼出し側から渡された実パラメータは書き換えられてしまいます。

１９－２．値を返すメソッド

１９－２－１．単一の値を返すメソッド

パラメータは、呼出し側からメソッドへのパラメータの受渡しでした。メソッドの実行の終了時には、メソッド側から呼出し側へ、何らかの情報を返すことができます。

★メソッドの終了と戻り値

メソッドの終了は、メソッドを定義しているブロックの最後まで実行が終わってしまえば、自動的に処理が呼出し側に戻ります。しかし、明示的にメソッドを終了したい場合は、**return**文を用います。これは、第１０章で出てきました。メソッドをどの時点でも強制終了させて、呼出し側に制御を戻せます。加えて、値を返すメソッドを作成したい場合は、呼出し側に値を返すときにも、この**return**文を用います。このときは、次の書式のように、returnの後に式を記述することができます。

▼return文で値を返すときの書式

return　メソッドを呼び出した方に返す値 ;

終了する前に、このreturn文の後に続く式が評価されます。評価が終わって一定の値になりましたら、呼出し側にその値が戻されます。ここで返される値のことを戻り値（return　value）あるいは返り値と呼んでいます。先ほどまでは、「返す値」として説明してきました。

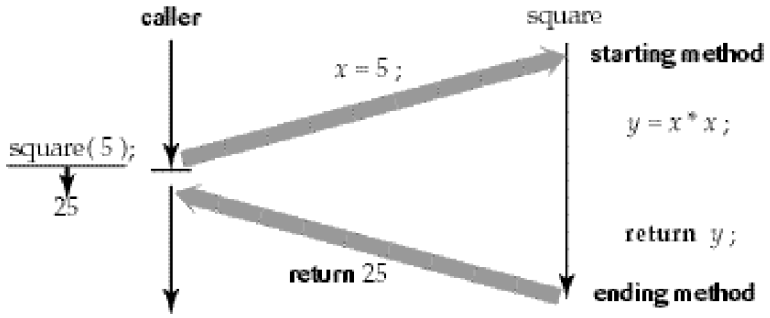

図19-5　return文による呼出し側への戻り値の返送

それでは、戻り値を返すようなメソッドを定義してみましょう。まず、戻り値を返すので、その型を記述します。今までは、戻り値がありませんでしたので、**void**と記述してきました。たとえば、整数を返すのであれば、**int**と記述します。簡単なメソッドを定義してみましょう。

```
int  square( int x ) {
        int y = x * x;
        return  y;
}
```

このsquareというメソッドはパラメータとして受け取った値の２乗を計算して、計算結果を戻り値として返します。整数用になっていますので、仮パラメータも戻り値も型としては**int**と記述されています。わかりやすくするために、ローカル変数*y*を用意して、それに計算結果を保持させるように記述しています。**return**の後は、式を書くことができますので、このメソッドは直接的に次のように定義することもできます。

```
int  square( int x ) {
        return  x * x;
}
```

もう１つの例を見てみましょう。２つのパラメータを受け取り、大きい方の数を返すメソッドを定義してみます。第１０章で出てきたif式を用いています。

```
int      greater( int x, int y ) {
        return  ( x > y ) ? x : y;
}
```

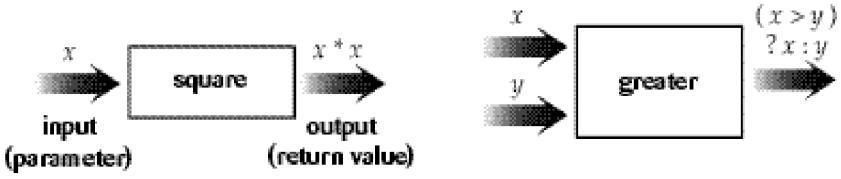

図19-6 squareとgreaterの機能図

戻り値は１つしか記述できないのでしょうか？C言語系統のプログラミング言語は、メソッドは元々は関数的

な取り扱いでしたので、戻り値は１つだけしか記述できません。もし、複数の値を戻したいのであれば、配列にしたり、オブジェクトの形に直す必要があります。その方法は後の章で説明します。

もし、戻り値を返す指定をした（たとえば、型を**void**ではなく**int**と記述した）のに、return文がメソッドの定義の中に含まれていなかった場合はどうなるでしょうか？大丈夫です。Java言語のコンパイラがちゃんと見張っていて、「return文がないよ」というエラーメッセージを出してくれます。

★メソッドの呼出しと戻り値の評価

では、戻り値のあるメソッドを呼び出して使ってみましょう。次のように、代入文の式の中にメソッド呼出しを記述します。実パラメータの個数は、定義された仮パラメータの個数と同じ分だけ用意します。

▼戻り値のあるメソッドの呼出し

```
戻り値を受け取る変数 = メソッドの名前( 実パラメータ );
```

もちろん、代入文など用意しないで、戻り値を受け取らなくても構いません。あるいは、メソッドの呼出しを複雑な式の中に埋没させても構いません。さて、それでは、先ほど定義したメソッドを、この書式に基づいて、呼び出して使うための記述をしてみましょう。変数を宣言しながら、代入を行なっています。

```
int     z = square( 30 );          // zには900が代入される
int     y = square( z + 20 );      // yには846400が代入される
int     w = greater( z, y );       // wにも846400が代入される
```

他の自作のメソッドと同様に、アプレット（クラス）自身のメソッドであることを示すためには、次のように**this**をつけて呼び出すこともできます。

```
int     positive = this.square( -30 );
```

このような呼出しを行なった場合、実際には、どのような実行の順番になっているのでしょうか？簡単な例を用いて、少し、実行の様子を追ってみましょう。

メソッドの呼出しの記述：

```
int     result = square( 45 * 10 );
```

→45*10という式が評価されて、定数値450になり実パラメータとしてsquareメソッドに渡されます。

メソッドの定義：

```
int  square( int x ) { return x * x ; }
```

→squareが呼び出され、仮パラメータ変数*x*に、実パラメータの定数値450が代入されます。
→*x* * *x*と書かれた式の計算結果の定数値202500が戻り値として呼出し側に返送されます

メソッドの呼出しの記述：

```
int     result = square( 45 * 10 );
```

→square(45 * 10)の部分が、戻り値の定数値202500に置き替わります。
→変数*result*を宣言し、*result* = 202500;という代入文が実行されます。

このように、実行の制御が一度定義されたメソッドに移り、計算が行なわれた後、再び呼出し側に戻ってきました。呼出し側では、メソッドを呼び出した部分の記述が、戻り値に置き換えられます。

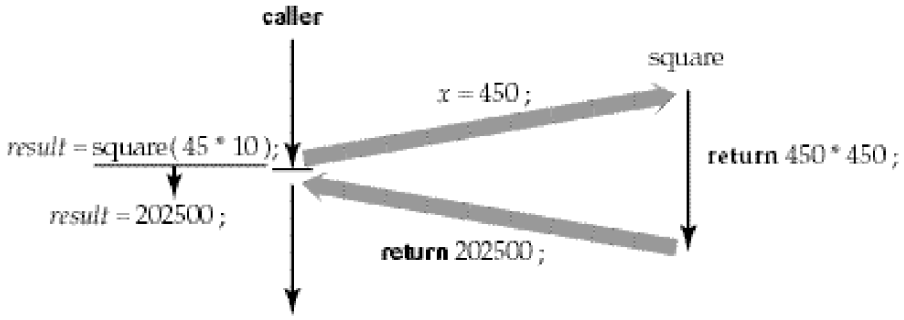

図19-7　squareメソッドの呼出しと戻り値の返送

★既出の戻り値のあるメソッド

今までも、戻り値のあるメソッドを使ってきました。どのようなものがあったでしょうか？少し、復習のために列挙してみます。もう、遠い過去のことかも知れませんが、ちょっと思い出してみてください。コンストラクタとして使われているメソッドは省いています。

何かの情報を得るメソッド：

getColor, getRed, getBlue, getGreen, getFont, getName, getSize, getStyle, getFontMetrics, getAscent, getDescent, getHeight, getLeading, stringWidth, getActionCommand, get, getID, getKeyChar, getKeyCode, getX, getY, getImage, getCodeBase, getDocumentBase, getAudioClip, getAppletContext, length, compareTo, subString, charAt, toString, valueOf, parseInt, intValue, floatValue, doubleValue, toCharArray, read, readLine, getText, getParameter

関数メソッド：

ceil , floor, rint, round, acos, asin, atan, atan2, sin, cos, tan, abs, exp, log, pow, random, sqrt

論理値を返すメソッド：

isShiftDown, isContolDown, isMetaDown, isAltDown, checkID, checkAll, waitForAll, equals, equalsIgnoreCase, endsWith, startsWith, ready

一杯使ってきました。そんなの使ったっけ？って思う人は、前の章を読み返してみましょう。どこかに書かれているはずです。情報を返すメソッドでは、getという動詞が、論理値を返すメソッドでは、isという動詞が多く名前の中に使われているのがわかります。自分で定義する際にも、メソッドに名前をつけるときにこのような慣習を守っておくと、後でプログラムを読み返したときにわかりやすいでしょう。

★複雑なメソッドの呼出し

メソッドの呼出しは、式の中に記述できますので、実パラメータを示す式の中にも別のメソッドの呼出しを記述することができます。次のような複雑なメソッド呼出しでは、内側の実パラメータから順番に評価されていきます。従って、呼出しの順番もより内側に書かれているメソッドから行われることになります。以下の記述では、greater→square→System.out.println と実行されることになります。

```
System.out.println( "result:" + square( greater( 30, 40 ) - 10 ) );
```

実際に、この１文がどのように実行されるかを追ってみましょう。

1. greater(30, 40)が実行される→評価結果は40になる
2. square(40 - 10)が実行される→評価結果は900になる
3. System.out.println("result: " + 900);が実行される

★論理値を返すメソッドの例

戻り値を返すメソッドの中でも、論理値を返すメソッドの場合は、if文やwhile文などの条件を記述する式に用いられます。たとえば、現在の時刻がお昼の12時から14時の間であることを判別するためのメソッドを記述し

てみます。

```
boolean  isLunchTime( )  {
        Calendar rightNow = new GregorianCalendar( TimeZone.getTimeZone("JST") );
        int      hour = rightNow.get( Calendar.HOUR_OF_DAY );
        return   hour > 12 && hour < 14;
}
```

論理値を返すので、戻り値の型としては**boolean**を指定しています。名前は、慣習に従って、動詞のisをつけています。Calendarクラスのオブジェクトを生成して、現在の時刻を求めています。日本なので、時差を考慮した形で記述しています。そこから、24時間として時刻を変数*hour*に求めています。**return**文の戻り値の式は、条件式になっています。このように記述すると、条件式が評価されて、論理値の**true**あるいは**false**が返されることになります。これを他のメソッドの中から呼び出して記述するようなことを考えてみましょう。

```
if ( isLunchTime( ) ) {
        System.out.println( "It is lunch time!  Shall we have lunch?" );
}
```

このように論理知を戻り値として返すメソッドを用いる場合は、メソッドの呼出しを直接**if**文の条件式の中に記述して使います。

１９－２－２．配列とメソッド

メソッドに配列を渡したり、配列を返したりしたい場合があります。その場合には、戻り値の型や、仮パラメータの引数の宣言のところに、配列であることを示す［　］を記述します。

```
例：    void scanArray( int a [ ] ) { ..... }          // 整数の配列を引数として貰う

例：    int [ ] getArray( ) { ..... }                  // 整数の配列を戻り値として返す
```

メソッドを呼び出す方では、実パラメータの中に配列名を記述します。

```
int       a [ ] = { 10, 20, 22, 5, 6 };      // ローカルな配列の定義
scanArray( a );                              // 配列aを引数として渡している
int       b [ ] =  getArray( );                    // 戻り値から、配列bを設定している
```

配列の場合は、配列の要素をメソッドの中から編集することができます。これは、オブジェクトを引数として与えるときも同じです。引数として与えられたオブジェクトの外部に公開された（publicがつけられた）メソッドやフィールドを呼び出されたメソッドの中で利用することができます。

配列を利用した例を考えてみましょう。例えば、次のメソッドは、論理値を返します。配列と、ある値が引数として与えられ、配列の要素がすべて、引数として与えられた値よりも大きかったら、trueを返します。そうでなければ、falseを返します。要素を調べていって１つでも小さい要素が見つかったらfalseを返して終了するようにしています。繰返しが最後まで終わったら、見つからなかったということでtrueを返しています。

```
boolean   isAllGreater( int  a [ ],  int  value ) {
        for ( int  i=0;  i< a.length; i++ ) {
                if ( a[ i ]  <  value ) { return  false; }
        }
        return  true;
}
```

これを利用してみましょう。

```
 int  array [ ] = { 18, 20, 40, 50, 30, 40 };
if ( isAllGreater( array, 15 ) ) {     ..... }          //  すべての値が15よりも大きかったら
```

次の例は、整数を返すメソッドですが、さきほどと同じように、配列と、ある値が引数として与えられ、もし配列の中にその値が存在したら、最初に見つかった要素のインデックスの値を返します。もしなければ、-1を返します。中身も、先程のメソッドと似ています。

```
int   scanValue( int  a [ ],  int  value ) {
        for ( int  i=0;  i< a.length; i++ ) {
                if ( a[  i ]  ==  value ) {  return  i ; }
        }
        return  -1;
}
```

これを利用してみましょう。

```
int  array [ ] = { 18, 20, 40, 50, 30, 40 };
int   target =  scanValue( array, 40 );
System.out.println(  "Target Index:" +  target );
```

１９－５．課題

19-1.
この章で出てきたいろいろな例題を完成させて、実際に動かしてみなさい。

19-2.
この章で定義した、scanInteger、displaySquare、displayCubeの３つのメソッドを用いて、ユーザが入力した整数の2乗、3乗を表示するようなアプリケーションプログラムを作りなさい。なお、ユーザが整数を表すような数字列を入力しなかった場合、間違っている！を表示して、もう一度入力させるようにしなさい。

19-3.
正多角形（三角形以上）を描画するようなメソッドを定義しなさい。パラメータとして受け取るのは、描画領域と、画数と、中心のx座標、y座標、および半径の長さの４つです。次のようなシグネチャになります。アプレットのプログラムを作り、このメソッドを用いて、正三角形～正２０角形までを描画してみなさい。

```
void  drawRegularPolygon(  Graphics g,  int n,  int x, int  y, int  radius  )
```

ヒント：360（2π）を画数で割ったりして角度差を求め、Math.sinメソッドとMath.cosメソッドおよび半径を用いて、各頂点の座標を割り出します。それらの座標間に線を引いていきます。あなたならできる！

19-4.
仮パラメータの３乗を戻り値として返す機能を持ったメソッドをcalculateCubeという名前を用いて、異なるシグネチャに対して定義しなさい。整数用のもの、実数用のもの、文字列用のものを定義しなさい。文字列のものは、仮パラメータで受け取った文字列を３つに繋げたものを返すようにします（それは３乗と違うような気がしますね）。

19-5.
整数の配列を受け取り、その配列の総和を返すようなメソッドを作りなさい。同じように、最大値を求めるメソッド、最小値を求めるメソッド、平均を求めるメソッドをそれぞれ作成しなさい。

19-6.
整数を受け取り、２進数として表現された文字列を返すようなメソッドを作りなさい。逆に２進数の文字列を受け取り、それを整数として解釈し、その整数を返すようなメソッドを作りなさい。