Chpater 17. 文字・文字列とテキスト入力

１７－１．文字型

１７－１－１．文字型の定数と変数

文字型は１つの文字を表現する型でした。文字定数の書き方は以前の章で一度説明しましたが、簡単に復習しておきましょう。そして、変数を使ったプログラミングに臨んでみます。

★定数の記述の仕方

文字型の定数はシングルクォーテーション（’）を使って囲みます。特殊コードの場合は、バックスラッシュ（\）を前において一字を記述します。Mac OS Xでは、バックスラッシュは、Option + ￥で出すことができます。

'A'　　→　　大文字のAを示します。
'鯖'　　→　　漢字の「鯖」を示します（注１）。
'\n'　　→　　改行コードを示します。

Unicodeを直接記述したいときはシングルクォーテーションで囲まれた中で\uに続けて4桁の16進数で表します。この4桁の16進数で文字の一文字が表現されます。

'\u4e00'　　→　　漢字の「一」（いち）を表します

★変数の記述の仕方

文字を表現する変数は次のように宣言、あるいは初期値代入することができます。

▼変数の宣言・初期値代入の書式

```
char   変数名;                          あるいは
char   変数名  =  文字定数;
```

例えば、次のように変数wを宣言したり、変数kanを宣言したりすることができます。変数*kan*の方には、初期値代入で文字が代入されています。

```
char    w;
char    kan = '漢';
```

★漢字とそのUnicodeの値を表示する。

簡単なアプリケーションを記述してみましょう。上の変数を使って、文字をそのまま表示し、加えてその文字のコードも表示させてみます。以下のプログラムのSystem.out.printlnメソッドの部分がその部分になっています。(int)の記述は、前に説明しましたが、文字を整数型に強制変換（Casting）させるものです。このようにすれば、整数として解釈されますので、文字コードが表示されることになります。

```
public  class  Kan {
        public  static  void  main( String [ ] args ) {
                char    kan='漢' ;
                System.out.println( "漢字の" + kan + "という字のUnicodeは" + (int)kan );
        }
}
```

初期値代入している部分を好きな漢字に替えて、Unicodeを表示させてみてください（注１）。あるいは、英数字に替えてみても構いません。上記のプログラムの場合には、端末上に次のように表示されます。

漢字の漢という字のUnicodeは28450

なお、整数値を16進数で表示することもできます。Integer.toHexStringメソッドを用います。これを使って、次のような記述することにより、16進数でUnicodeを表示させることもできます。

System.out.println("漢字の" + *kan* + "という字のUnicodeは\\u" + Integer.toHexString((**int**) *kan*));

この場合には次のように表示されます。

漢字の漢という字のUnicodeは\u6f22

※注１　JavaのソースプログラムがShift JISコードで書かれている場合は、Project Builderの設定でテキストエンコーディングを指定しておこないと、漢字で書かれた文字や文字列をUnicodeに変換してくれません。

１７－１－２．文字と演算

文字は整数のように比較したり、加減算を行なうことができます。ただし、加減算を行なった結果は、整数になってしまいますので、再度文字として扱うためには、強制的に文字型に再変換してやらなければなりません。このときの、型変換には(char)というCastingの記述を用います。

★比較

比較はUnicodeのコードの順番によって行なわれます。ですから、例えば、数字の0〜9までの文字は、次のような特性を持っています。￥

'0' ＜ '1' ＜ '2' ＜ '3' ＜ '4' ＜ '5' ＜ '6' ＜ '7' ＜ '8' ＜'9'

同じように英字もAからZまで順序付けられています。漢字の場合は、一概には言えませんが、漢和辞典に載っているように偏（へん）の画数で大きな順序付けがされており、同じ偏の漢字では旁（つくり）の画数で順序付けられています。カタカナ、ひらがなは、「あいうえお」順で順序付けがされています。

★１つ前と１つ後

Unicodeは、整数ですので、整数としての演算はすべて使えるのですが、特にコードが連続している数字や英字などでは、順序性を利用して、文字の１つ前と１つ後などを計算することがあります。例えば、大文字のMの１つ前と１つ後を求めてみましょう。次のように記述すれば、NとLが求められます。

(**char**) ('M' + 1) → 'N'
(**char**) ('M' - 1) → 'L'

★比較と演算の典型的な例

以下の３つのif文では、典型的な文字の使い方を示しています。前提として、次のような変数が宣言されているとします。

```
char    c, w;                                   // cに何らかの文字が保持されているとする
int     digit ;
```

変数に何がはいっているかを確かめるには次のように定数と比較します。このように、大文字でも小文字でも構わないとときには、論理和を用います。

```
if ( c == 'A' || c == 'a' ) {                   // cと文字定数を比較
```

```
            System.out.print( "Aだ！");
        }
```

変数に入っている文字が、英大文字かどうか比較するときは、次のようにします。この例では、更に小文字に変換して別の変数に代入しています。

```
        if ( 'A' <= c  &&  c <= 'Z' ) {                    // cが保持する文字が大文字かどうか判定
            w = (char) (c + 32);                           // 小文字にしてwに代入
        }
```

次の例は、数字かどうかを確かめています。数字であれば、一桁の10進数に変換しています。文字の'0'を引くことによって、差分を求めることができます。その差分が、実際の数値になります。例えば、'5'の場合には、'5' - '0' → 53 - 48 → 5ということになります。Unicodeのコード表で確かめてください。

```
        if ( '0' <= c  &&  c <= '9' ) {                    // cが保持する文字が数字かどうか判定
            digit = c - '0';                               // 一桁の10進数に変換
        }
```

１７－１－３．コード表を表示する。

Javaの実行環境に、コードに対応した字体が用意されていなければ、文字は表示されないのですが、範囲を指定して、コードと文字を表示するようなプログラムを記述してみましょう。アプリケーション用とアプレット用の両方を記述してみます。

★アプリケーション版
アプリケーション晩では、文字型を使って繰返しを行なっています。Unicodeとして、\u0020〜\u8000までの範囲のコードの文字を表示しています。16個表示するごとに、コードを16進数の整数で表示しています。

```
public class GeneralScripts {
    public static void main( String [ ] args ) {
        for ( char code='\u0020' ; code < '\u8000' ; code++ ) {
            if ( code % 16 == 0 ) {
                System.out.print( "\n" + Integer.toHexString( (int) code ) );
            }
            System.out.print( "    " + code );
        }
    }
}
```

★アプレット版
アプレット版では、文字コードを整数型にして繰返しを行なっています。Unicodeとして、\u4e00〜\u4fffまでの範囲のコードの文字を表示しています。16個表示するごとに、コードを16進数の整数で表示しています。

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

public class LocalScripts extends Applet {
    public void paint( Graphics gc ) {
        int     base = 0x4e00;                         // この数値を変えてみなさい

        for ( int code=base ; code < base + 0x1ff ; code++ ) {
            if ( code % 16 == 0 ) {
                gc.drawString( Integer.toHexString( code ), 40, (code-base)/16 * 20 + 30 );
            }
            gc.drawString( ""+(char)code, code%16 * 20 + 40, (code–base)/16 * 20 + 30 );
        }
    }
}
```

前にやった整数の剰余と、整数除算を使って、表示する位置を調整しているのに注意してください。

１７－１－４．日本語の扱い

Mac OS Xでは、日本語を扱うのに、主に２つのコード体系を使っています。標準では、Shift-JISコードが設定されています。また、Unicodeもアプリケーションなどでは扱っています。しかし、Javaの標準の漢字コード体系は、Unicodeになっています。特に、Xcodeなどで、プログラム中に漢字を使った場合、うまく表示されない場合があります。そこで、漢字などを取り扱う場合は、Xcodeでは以下のような設定をしてください。

１．「ターゲット」の垂直タブを開きます。
２．一番上の方に出ているプロジェクト名と同じ名前のターゲットをクリックします。
３．クリックしたら表示される「設定」の中から、「Javaのコンパイル設定」を選びます。
４．「ソースファイルのエンコーディング」を「Unicode（UTF-8）」に設定します。
５．この時点で、「ファイル」メニューから「保存」を選び、一旦設定を保存します。

１７－２．文字列

１７－２－１．文字列の定数と変数

複数の文字を表すのは文字列を使います。文字列定数の書き方は以前に記しましたが、ここでも簡単に復習しておきましょう。そして、オブジェクトとして定義されている文字列を扱う変数を導入します。

★文字列定数の書き方

文字列定数はダブルクォーテーション（"）で囲みます。文字定数で用いることのできる制御コードの記述（バックスラッシュ\を用いる）方法やUnicodeを16進数で記述する（\uで始める）方法も同じように用いることができます。次の文字列は、その両方の記述方法を含んだ文字列です。

```
"千里の路も\u4e00歩から\n"
```

★文字列変数の使用方法

これまでのプログラミング言語では（例えばCやPascalなど）、文字列を文字型の配列を使って表現する方法が一般的でした。しかし、Java言語では、文字配列は用いる代わりに、文字列を表すためにStringクラス（文字列クラス）が特別に用意されています。このクラスの変数のを宣言するのは、通常のオブジェクトを参照する変数を宣言すると同じで、次のように記述します。もちろん、初期値代入も可能ですので、その場合は２番目の記述の仕方を用います。

▼文字列変数の宣言の書式

```
String   変数名;                          あるいは
String   変数名  = 文字列定数;
```

通常のオブジェクトを参照する変数よりも少し使い勝手が良い部分があります。２番目の初期値代入で、文字列定数をそのまま記述すれば、その文字列定数を参照することができるからです。

```
String  wow;
String  please = "お願い、許して！";
```

宣言した文字列変数は、次の例のように、プログラム中の文字列の定数が記述できる場所に、文字列変数を記述することができます。

```
wow =  "鱈子が食べたい" ;
System.out.println(  please  );
```

```
gc.drawString( wow, 100, 20 );
```

★文字列の参照の変更といらなくなった文字列の廃棄

代入文を用いて、同一の文字列変数に次々と文字列定数を代入していった場合、前に代入されていた文字列は参照されなくなります。

```
String    fickle = "最初の人" ;
fickle = "次の人" ;
fickle = "更に次の人" ;
```

参照されなくなった文字列定数は、参照されなくなったオブジェクトと同様にいずれガベージコレクションの機構によって、メモリ上から廃棄されることになります。

１７－２－２．文字列の結合

加算演算子+が文字列と文字列を結合するのに、あるいは文字列と別の型のデータを結合するのに用いられます。結合された結果は、すべて文字列になります。次の例は、文字列と文字列を足しあわせて文字列変数に代入したものと、文字列と別の型の定数（文字や整数、実数など）を足しあわせて文字列変数に代入したものでうす。最後に文字列変数どうしも足しあわせています。

```
String    combined = "おせちもいいけど"  + "カレーもね" ;
String    complicated = "文字と”  +  '数' + 30 + "や実数" + 4.5e3 + "を結合" ;
System.out.println( combined + complicated );
```

文字列変数に対して自己再帰的なの加算演算もできます。以下の例では、変数*extended*は毎行異なる文字列を指していくことを注意してください。この例では、結合の計算結果として発生する新しい文字列を参照していくことために、前に代入された文字列は参照されなくなります。

```
String    extended = "うらら";
extended = extended + "うらら";        // "うららうらら" を新しく参照する
extended += "うらうらら";              // "うららうららうらうらら" を新しく参照する
```

文字列の演算で有効なものは、この加算演算しかありません。減算や乗算などの演算子はありませんし、==や>などの比較演算も、オブジェクトの識別子の比較になってしまうので、意味を持ちません。文字列の一部を取り出したり、文字列どうしを比較するためには、Stringクラスに用意されているいくつかのメソッドを利用します。

１７－２－３．文字列のサイズを知るには？

文字列はStringクラスのオブジェクトですので、メソッドを持つことが可能で、lengthというメソッドが用意されています。このメソッドを使って初期値代入された文字列などのサイズを知ることができます。注意しなければならないのは、配列の場合は属性であったので、lengthで良かったのですが、文字列の場合はメソッドなので、length( ) と括弧を付けなければならない点です。

▼サイズを知るための式

```
文字列の定数や変数 .length( )
```

たとえば、次のように文字列変数に定数を代入して、文字数を得ることができます。最初の方のlengthメソッドの呼出しでは、文字列が17文字から構成されていますので、変数*lengthofmaki*には17が代入されます。この例では、わかりやすくするために、一旦整数の変数にサイズを代入していますが、直接maki.length()をその下の行のSystem.out.printlnの引数の中に入れても構いません。また、その下の例のように、文字列定数に対して

直接lengthメソッドを呼び出すことができます。この場合は、6という結果が返ってきます。

```
String   maki= "ああ、やんなっちゃう、ああ、驚いた";
int      lengthofmaki =  maki.length( );
System.out.println(  "文字変数makiの中の文字数は" + lengthofmaki );
System.out.println(  "この文字数は" + "この文字数は".length( ) );
```

１７－２－４．文字列の比較

オブジェクトに対して、等しいという意味の==と、等しくないという意味の!=を頻繁に用いることができます。しかし、これはオブジェクトの同一性（Identification）をテストするためであって、文字列として等価である（Equivalence）ということを比較することはできません。等価性を確かめたい場合には、比較演算子の替わりに、Stringクラスに用意されている次のようなメソッドを用います。

equals(文字列)	等しいかどうか
equalsIgnoreCase(文字列)	等しいかどうか（大文字小文字の別は無視する）
endsWith(文字列)	その文字列で終了するかどうか
startsWith(文字列)	その文字列で開始するかどうか

これらのメソッドは、満足する場合は、**true**を返してきます。満足しない場合は、**false**を返してきます。各メソッドを用いてみましょう。

★等しいかどうかテストする

次の例は、文字列変数planeに代入されている文字列変数が特定の文字列と等しいかどうかif文を用いてテストしています。

```
String    plane  =  "F16 Falcon";

if  ( plane.equals( "F16 Falcon"  ) ) { System.out.println( "Air Force Fighter" ); }
else  if  ( plane.equals(  "F18 Hornet"  ) ) {  System.out.println( "Navy Fighter" );  }
else { System.out.println( "Is this a plane of US ?" ); }
```

このequalsメソッドは、アプレットで複数ボタンが用意されているときに、どのボタンが押されたか判別するために、アクションコマンドをボタンに設定して置いた例で、アクションコマンドの文字列を比較するために用いられています。

★大文字小文字の別は無視して、等しいかどうか

equalsIgnoreCaseを利用しますと、文字列中の大文字、小文字の区別は無視してくれます。次の例は、if文の中に現れたこのメソッドのパラメータの中では、比較する文字列は大文字で書かれていますが、文字列変数が保持する小文字混じりの文字列と等しいと判断されます。

```
String    mytext = "Igor Stranvinsky";

if  ( mytext.equalsIgnoreCase( "IGOR STRAVINSKY" ) ) {
        System.out.println(  mytext + " composed the Fire Bird." );      }
```

★指定された文字列で開始・終了するかどうか

次の例は、それぞれのif文の中で、文字列がplaying?で終わるかどうか、Whenで始まるかどうか調べています。最初のif文の条件式は、**false**に評価されますし、次の条件式は**true**に評価されます。

```
String    message = "When did you stop eating?";

if  (  message.endsWith( "playing?" ) )                          //        false
        { System.out.println( "Oh, I used to think I would like to study anytime." ); }
```

```
if ( message.startsWith( "When" ) )                    //        true
        { System .out.println( "I used to stop everything before noon" ); }
```

★辞書順の順序を比較する

等しいことを確かめるのに==演算子が使えないのと同様に、文字列を辞書順に比較するためには、<や>=などの比較演算子は利用することはできません。次のcompareToというメソッドを利用します。

compareTo(文字列)　　文字列の辞書順を指定した文字列と比べる

このメソッドは、整数を返してきます。指定した文字列よりも辞書順で先であれば負の数を返してきます。また、等しい文字列なら0を返してきます。辞書順で後になるならば、正の数を返してきます。

```
String    source = "This is a small message.";

int result1 = source.compareTo( "This" );                          //      結果は正の数
int result2 = source.compareTo( "No, it's long." );                //      結果は正の数
int result3 = source.compareTo( "This is a small message." );      //      結果は0
int result4 = source.compareTo( "This is not a small message." );  //      結果は負の数
int result5 = source.compareTo( "What is the message?" );          //      結果は負の数
```

負の数、正の数が一体どの数になるかは決まっていません。ですから、if文などで用いるときは、次のように0との不等号で判別する必要があります。

```
String    target = "Rabbit" ;
if ( target.compareTo( "Lion" ) > 0 ) {      System.out.println( "Lion proceeds the target" ); }
if ( target.compareTo( "Turtle" ) < 0 ) {    System.out.println( "Turtle follows the target" ); }
```

１７－２－５．文字列の一部分を取り出す

★位置を指定して一部分を取り出す

文字列の先頭何文字かだけを必要としたり、途中の何文字かだけを必要する場合があると思います。文字列の一部を取り出して、別の文字列として作成して返すメソッドとして、substringメソッドが用意されています。

文字列.substring(開始位置, 終了直後の位置)　あるいは
文字列.substring(開始位置)

開始位置（0〜length( ) -1の範囲）から、終了位置（0〜length( )の範囲）の手前までの文字列を取り出してくれます。取り出すのが終了直後の位置の「１文字分手前」であるということに注意してください。終了直後の位置を省略すると、開始位置から文字列の最後までを取り出してくれます。もちろん、取り出した後の文字列は別の文字列として生成されますので、元の文字列は変わらないで残されます。

次の例は、3文字目から9文字目を取り出したものと、7文字目以降最後までを取り出したものです。

```
String   message = "Sample Message";
System.out.println( message.substring( 3, 10 ) );        // "ple Mes"
System.out.println( message.substring( 7 ) );            // "Message"
```

先頭の5文字分だけ取り出したいとき、あるいは最後の5文字分だけ取り出したいときは、それぞれ次のように指定します。最後の何文字かだけ取り出したいときは、パラメータが１つだけの方のsubstringを使い、lengthメソッドと用いてサイズを求め、そこから文字数分を引くようにして、開始位置を計算しています。

```
String    firstFive = message.substring( 0, 5 );                  // "Sampl"   最初の５文字
String    lastFive = message.substring( message.length( ) - 5 );  // "ssage"   最後の５文字
```

１７－２－６．文字列中の個々の文字を参照したい

文字列では配列と同様に文字列の各文字を参照することができます。位置は配列と同様に0から始まり、文字列のサイズ-1までの範囲で指定することができます。参照には、charAtと呼ばれるメソッドを利用します。このメソッドは、文字列中から指定した位置の文字を返してくれます。

▼位置を指定して文字を参照するための式

```
文字列の定数や変数.charAt( 最初からの位置 )
```

たとえば、次のようにすれば、それぞれ、先頭の文字、最後の文字を求めることができます。配列のインデックスと同じで、位置は0から始まることに注意してください。

```
Srting   crazyCats = "わかっちゃいるけど、やめられない";
char     firstchar = crazyCats.charAt( 0 );                    //      'わ'
char     lastchar = crazyCats.charAt( crazyCats.length( ) - 1 );   //      'い'
```

charAtメソッドとsubstringメソッドとの違いに注意してください。substringメソッドでも１文字を取り出すことができますが、取り出した結果は、文字列です。charAtメソッドの場合は、取り出した結果は文字になっています。

```
String   katochan = "ちょっとだけよ" ;
String   fourth = katochan.substring( 3, 4 );          // "と" ←文字列
char     fourthchar = katochan.charAt( 3 );            // 'と'  ←文字
```

★初期値代入されたすべての文字列を１文字ずつ表示する

繰返しとcharAtメソッドを利用すれば、次のようにして、１文字ずつ順番に取り出していくことができます。

```
String   takostr = " 蛸はどうした？";
for ( int  i =0; i < takostr.length( ); i++ ) {
        System.out.println( i + "番目の文字は" + takostr.charAt( i ) + "です。" );
}
```

★単語がいくつあるか数える

単語と単語の間は１つ空白があり、文字列が空白で始まったり、空白で終わったりすることがないという簡単な場合だけを考えてみましょう。変数wordcountで単語の個数を数えます。空白で始まることがありませんから、必ず１語はあるということになります。そして、文字列を先頭から順に見ていき、空白が出てくる度にwordcountの値を１ずつ増やしていきます。文字列の最後まで見終わったときのwordcountの値が単語数になっています。

```
String   message = "This is a sample message";
int      wordcount = 1;
for ( int  i=0;  i  < message.length( );  i++ ) {
        if  (  message.charAt( i ) == 32  ) {     wordcount++; }    //空白が現れたら単語数を増やす
}
System.out.println(  message + "の単語の数は" + wordcount );
```

上の例で、if文の中で32と比較しているのは、空白のUnicodeでの文字コードは32だからです。この部分を、次のように空白一個分を開けた文字定数を用いて記述しても構いません。

```
if  (  message.charAt( i ) == ' '  ) {    wordcount++; }
```

１７－２－７．文字単位で既存の文字列を変更する

これはちょっと面倒です。Stringクラスは、後で文字を変更するために用意されたクラスではないので、そのままでは既存の文字列を変更することができません。次のようにsubstringと結合演算子（+）を用いれば、一部分を変更することはできますが、これでは新しい文字列を生成してしまいますし、記述も面倒ですし、あまり効果的ではありません。

```
String    original =  "私は鮪を食べたいのです。" ;
String    modified = original.substring( 0,  2 ) + '鰹' + original.substring( 3 );
                                        // →"私は鰹を食べたいのです。"という文字列が新しく生成される
```

このように、Stringクラスは、一定の文字列を効率的に処理をするために設計されており、既存の文字列を書き換えることを想定していません。そのため、文字列の個々の文字を頻繁に変更するような場合は、非効率的になります。そのような用途には、Stringクラスのサブクラスであ るStringBufferクラスを用います。

★Stringクラスの代わりにStringBufferクラスを用いる

StringBufferクラスは、Stringクラスと異なり、同一のメモリ領域にある文字列に対して操作していくための文字列クラスになっています。StringBufferクラスを用いる場合は、通常のオブジェクトの用に変数を宣言してそれに文字列を代入してから使うようにします。

```
StringBuffer        buffer =  "A sample message";
```

上の例のように、StringBufferクラスの変数は、変数の宣言のときにStringとするところをStringBufferに置き換えるだけ、あとは通常のStringクラスの変数のように用いることができます。しかし、それに加えてStringBufferクラスのオブジェクトには、文字単位で文字列を変更するためのsetCharAtメソッドが用意されています。以下のように使います。

▼文字単位で変更する書式

```
変数名.setCharAt(  文字の位置 ,  変える文字   );
```

文字の位置は、配列と同様に0〜文字列のサイズ-1の間で指定することができます。このメソッドを用いて変数に代入されている文字列を１文字単位で変更させることができます。文字単位で頻繁に文字の変更がある場合は、StringBufferクラスを用いた方がよいでしょう。StringBufferクラスはStringクラスのサブクラスですので、Stringクラスのオブジェクトで利用できるすべてのメソッドが利用可能になっています。

```
StringBuffer   takostr = "蛸はどうした？";
takostr.setCharAt(  0,  '鮪'  );                    // "鮪はどうした？" に変更されます
System.out.println(  takostr );
```

１７－２－８．文字列用の更に便利なメソッド

いくつかの便利なメソッドが用意されています。それを少しだけ御紹介しましょう。

★indexOf(探したい文字   )

その文字が何文字目から始まるか、教えてくれます。最初の文字の位置はは０から始まることに注意してください。加えて、同じ文字をそれ以降に探したいときは、つぎの２つのパラメータを持つ同じ名前のメソッドを使います。なお、両方のメソッドとも、探したい文字がない場合は、-1を返してきます。

```
indexOf(  探したい文字,  探し始める位置 );
```

実際の例を見てみましょう。次のプログラムの断片は、「桃」という字を２回探しています。

```
String   message  = "桃も李も桃のうち";
int      search =  message.indexOf(  '桃' );                    // 結果は0
int      search2 = message.indexOf( '桃', search+1 );     // 結果は4
```

★replace(　古い文字，新しい文字　)

指定した文字をすべて取り替えてくれます。実際の例を見てみましょう。次のプログラムの断片は、「桃」を「蛸」に置き換えています。

```
String   message = "桃も李も桃のうち";
String   mymessage = message.replace( '桃', '蛸' );              // "蛸も李も蛸のうち"
```

下に示しましたStringBufferクラスに用意されている同じ名前のメソッドは、３つのパラメータを取りますが、こちらは、指定した範囲にある部分文字列を、指定した文字列で取り替えてくれます。取り替える文字列の長さが、指定した範囲よりも長い場合は、指定した範囲の長さ分だけしか、取り替えてくれません。

replace(　先頭の位置，終了直後の位置，取り替える文字列)

実際の例を見てみます。次の例は、「さんま」を「たらこ」に取り替えています。

```
String   sentence = "目黒のさんまが食べたいなあ";
sentence.replace(  3,  6,  " たらこ");                    //  "目黒のたらこがたべたいなあ"
```

★toLowerCase( )およびtoUpperCase( )

文字列のすべての文字を小文字、あるいは大文字に変えてくれます。次の例は、すべて小文字で表示し、その後にすべて大文字で表示します。

```
String message= "This is a simple message";
System.out.println(  message.toLowerCase( ) );          // "this is a simple message"
System.out.println( message.toUpperCase( ) );           // "THIS IS A SIMPLE MESSAGE"
```

★trim( )

文字列の先頭や最後にある余分な空白を除去してくれます。

```
String   whiteSpace = "     Snow in the north field.      ";
System.out.println( whiteSpace.trim(  )  );             //  "Snow in the north field"
```

１７－２－９．文字列と他の型とのデータ変換

★文字列型への変換

整数、文字、実数などの基本型を文字列に変換する方法は、それぞれの基本型に対応するクラス（ラッパークラス：Wrapper Classと呼ばれています）に用意されているtoStringメソッドを呼び出すことによって実現できます。次は、整数型の定数30と、実数型の定数3.9、および文字型の定数「鰹」を文字列型に変換してみた例です。

```
(new Integer(30)).toString( )          →      "30"に変換される
(new Double(3.9)).toString( )          →      "3.9"に変換される
(new Character( '鰹' )).toString( )    →      "鰹"に変換される
```

あるいは、文字列に変換するためには、Stringクラスに用意されているvalueOfメソッドを用いることもできます。こちらの方が簡単でしょう。

```
String.valueOf( true );                    //    "true"
String.valueOf( 12 );                      //    "12"
String.valueOf( 12.34 );                   //    "12.34"
String.valueOf( 'A' );                     //    "A"
```

しかしながら、手っ取り早く文字列に変換するには、文字列の結合演算子を使って、空文字列である""と結合するのがいいでしょう。この結合演算子は、内部的にはvalueOfメソッドやtoStringメソッドを使って実現されています。

```
"" + 30          →      " 30"に変換される
```

★文字列からの変換

整数を表現する文字列を整数値に変換するにはIntegerクラスにメソッドが用意されています。

▼数字列を整数に変換する式

```
Integer.parseInt(   文字列   )
```

たとえば、次の記述は、文字列の"39"を整数を示した数値列として解釈して、整数値の39に変換して、変数に代入しています。

```
int          thirtynine =  Integer.parseInt(  "39" );
```

Java2からは、数字列を実数に変換するメソッドがDoubleクラスに追加されました。

▼数字列を倍精度実数に変換する式

```
Double.parseDouble(   文字列   )
```

たとえば、次の記述は、文字列の"-123e4"を実数を示した数値列として解釈して、実数値の-123×10^4に変換して、変数に代入しています。

```
double       onetwothree =  Double.parseDouble(  "-12.3e4" );
```

同様に、Float.parseFloatメソッドも用意されています。

▼数字列を単精度実数に変換する式

```
Float.parseFloat(   文字列   )
```

以下のような形で使います。

```
float        fvalue =  Float.parseFloat(  "-12.3" );
```

JDK1.1までの環境では、実数の場合は、次のように基本型に対応するラッパークラスに用意されているコンストラクタメソッドを用いて、ラッパークラスに変換しました。そして、そのクラスに用意されている基本型に変換するメソッドを呼びます。整数型のもの含めて御紹介しましょう。

```
int       i = (new Integer( "45" )).intValue( );
float     f = (new Float( "34.5" )).floatValue( );
double    d = (new Double( "35.2e-3" )).doubleValue( );
```

上の例では、文字列の"45"を整数値に変換して変数*i*に、文字列の"34.5"を単精度実数に変換して変数*f*に、文字列の"35.2e-3"を倍精度実数に変換して変数*d*に代入しています。

１７－３．GUIを利用した文字列入力

Javaではユーザからの入力を得るためには、マウスやキーボードからの低レベルのイベント処理によって行なうか、ボタンなどの構造物（コンポーネント）を利用する方法を採ります。また、ユーザに特定の文字列を入力してもらい、その入力文字列をプログラム中に取り込むためには、テキスト入力用のコンポーネントを用意する必要があります。ここでは、テキスト入力用のコンポーネントを使った文字列入力の方法について説明します。

１７－３－１．文字列の入力用のコンポーネント

Buttonクラスと同じように、文字列を入力したり表示したりするためのコンポーネントを用意します。このコンポーネントを表現するオブジェクトを作るためには、TextFieldクラスかTextAreaクラスを使うことになります。これらのクラスは、TextComponentクラスのサブクラスになっています。これらのクラスのオブジェクト（以降テキストフィールドとテキストエリアと呼びます）を生成するときにはコンストラクタでそれぞれ次のように指定します。

```
new TextField( カラム数　)
new TextArea( 行数，カラム数　)
```

テキストフィールドは１行だけですが、テキストエリアは複数行編集することができます。行数やカラム数は整数式で指定します。これらのクラスには、文字列を設定するためのsetTextメソッド、あるいは入力された文字列を受け取るためのgetTextメソッドが用意されています。このコンポーネントを配置してしまえば、ユーザとのこまごまとした入力のやり取りは、AWT パッケージが自動的にやってくれます。

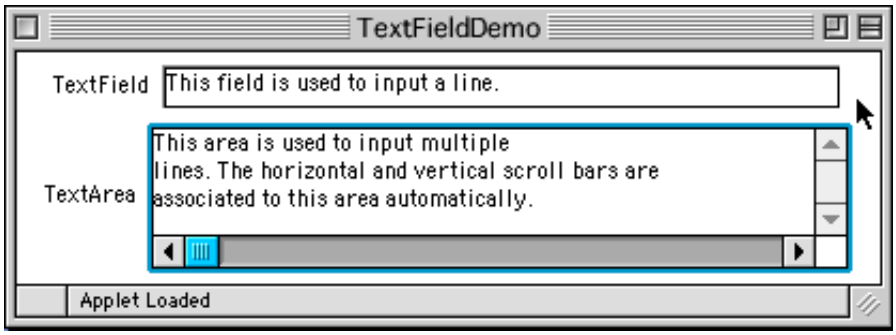

図17-1　テキストフィールドとテキストエリア

また、既存のテキストに追加するのは、appendメソッドが用意されています。appendメソッドは繰返しなどで多用しますと動作が遅くなることもありますので、注意してください。そのときには、文字列を結合しておいて、最後に１回、setTextメソッドか、appendメソッドで設定した方が良いでしょう。

★文字列を受け取るための手順

たとえば、アプレット用のプログラムを作る場合は、次のような形で入力を受け取るようにします。

１．入力を受け取るテキストフィールドあるいはテキストエリアを設けて、配置しておきます。これは、initメソッドの中で行ないます。setTextメソッドを使って、予め何か入力してもらうようにメッセージを表示しておくのもよいでしょう。

```
TextField  tf = new TextField( 30  ) ;
tf.setText(  "Please input here." );
add( tf );
```

２．ボタンなどを設けておいて、押されたときに、テキストフィールド（テキストエリア）からgetText()で入力された文字列を受け取るようにします。これは、ボタンを押されたときに実行されるactionPerformedメソッドの中で実行することになるでしょう。

```
String   received  =  tf.getText( );
```

３．入力されたテキストにさらに付け加えたいときは、appendメソッドを用います。

```
tf.append(  ": This is your input." );
```

★入力された内容を再表示するアプレットの例：

次のプログラムは、テキストエリアとボタンを用意したアプレットです。ボタンがおされたときに、テキストエリアから文字を受け取って、前後にメッセージをつけて再表示させています。今まで出てきたアプレットとは異なり、paintメソッドを定義していません。コンポーネントの再描画は自動的に行なわれますし、それ以外に何かグラフィックスを表示していないからです。

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

public class TextInput extends Applet  implements  ActionListener {
        TextArea        tarea;
        Button          button;

        public void init( ) {
                tarea = new TextArea(  5, 40  );
                button = new Button( "OK" );
                button.addActionListener(  this );
                add( tarea );
                add(  button );
        }

        public void actionPerformed( ActionEvent e  ) {
                String received = tarea.getText( );
                tarea.setText( "入力されたものは、"+ received + "ですね" );
                repaint( );
        }
}
```

図17-2　TextInputアプレットの実行例

１７－３－２．テキストフィールドで直接応答するメソッドを起動する

TextFieldクラスのオブジェクトの場合はTextAreaクラスとは少し異なり、ボタンを用意しなくても、アクションに対して応答するさせることができます。テキストフィールドにアクションリスナーを直接付加することができるからです。テキストフィールドでは、通常ユーザが改行コード（ReturnやEnterなど）を入力したら、アクションイベントが発生します。

次のアプレットのプログラムは、２つのテキストフィールドを用意しています。最初のテキストフィールドの方に英語の単語の単数形を入力します。このテキストフィールドには、アクションリスナーを付加しています。アプレット自体がリスナーになるように指定しています。このテキストフィールドでユーザが単語を入力したら、actionPerformedメソッドが呼び出され、その中で単語の終了文字をチェックしています。単語が、s, x, ch, あるいはshで終わっていたら、複数形をesにして２つ目のテキストフィールドに単語を設定しています。単語がyで終わっている場合は、yを取り除いてiesをつけています。それ以外の場合は、sをつけています。

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

public class WordPlural extends Applet  implements  ActionListener {
        TextField          singular, plural;

        public void init( ) {
                singular = new TextField( 20 );
                plural = new TextField( 20 );
                singular.addActionListener(  this );
                add( singular );
                add( plural );
        }

        public void actionPerformed( ActionEvent  e  ) {
                String word = singular.getText( );
                if ( word.endsWith( "s" ) || word.endsWith( "x" ) ||
                    word.endsWith( "ch" ) || word.endsWith( "sh" ) ) {
                        plural.setText( word + "es" );
                } else if ( word.endsWith( "y" ) ) {
                        plural.setText( word.substring( 0, word.length(  )-1 ) + "ies" );
                } else {
                        plural.setText( word + "s" );
                }
                repaint( );
        }
}
```

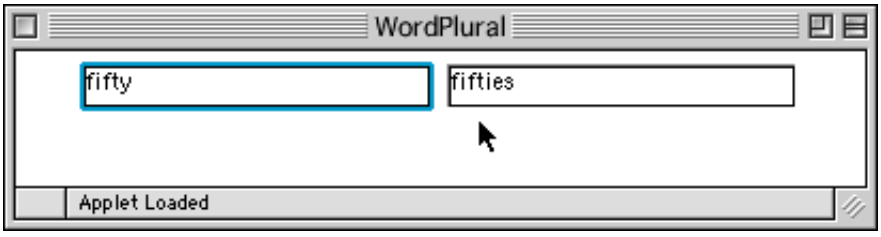

図17-3　WordPluralアプレットの実行例

１７－３－４．システムのプロパティを取り出す

TextAreaでは、改行コードなどが使えます。しかし、改行コードについては、各オペレーティング・システムでコードが違っていますので、その実行環境にあわせたコードを取り出す必要があります。このように、実行環境に応じた様々な特性を取り出すために、SystemクラスにgetPropertyメソッドが用意されています。このメソッドは、文字列を返してきますので、その文字列を使うことができます。また、特性を指定するの

も、文字列として指定します。特性には、次のような文字列を指定します。

システムプロパティ名	意味
java.vendor	ベンダ固有の文字列
java.vendor.url	ベンダのURL
java.version	Javaのバージョン
java.home	Javaの導入ディレクトリ
java.class.version	Javaのクラスバージョン
java.class.path	クラスパス
os.name	オペレーティング・システムの名前
os.arch	オペレーティング・システムのアーキテクチャ名
os.version	オペレーティング・システムのバージョン
file.separator	ファイルの分離記号（例：￥や／など）
path.separator	パスの分離記号（例：；や：など）
line.separator	改行記号
user.name	ユーザの名前
user.home	ユーザのホームディレクトリ
user.dir	ユーザの作業ディレクトリ

たとえば、オペレーティング・システムの名前を取得するためには、次の様に記述します。

```
String   name = System.getProperty( "os.name" );
```

改行コードを取り出すのは、次のようにして行ないます

```
String   lsep  = System.getProperty( "line.separtor" );
```

なお、システムプロパティの文字列を間違えると、コンパイラはエラーはまったく出しませんが、実行時にセキュリティエラーが発生します。実行時にセキュリティエラーが出た場合は、上記のプロパティの綴りが間違っていないかどうか、もう一度確かめてください。

１７－３－５．数値データの入力

Javaには、データを入力としてプログラムに取り込む際には、文字あるいは文字列として取り込むことしかできません。数字やピリオドなどから構成される文字列を数値、整数や実数として解釈するためには、文字列からそのような型のデータ値に変換する必要があります。よく誤解されるのですが、Javaにはデータストリーム（DataStreamクラスのオブジェクト）と呼ばれるストリームが用意されているのですが、これは整数や実数のメモリ上でのイメージをそのまま入出力するためのストリームで、数字列を整数や実数として解釈してくれるものではありません。

★数値の入力の変換と受取り

ユーザから数値の入力を受け取るために必要なことを復習として整理してみましょう。第一に、文字列から整数や実数に変換するのは、次のような記述をする必要がありました。下の記述のうち、最初の方は"39"という文字列を整数の39に変換していますし、次の方は"23.45"という文字列を実数の23.45に変換して変数に代入しています。

```
int      w =  Integer.parseInt(  "39" );
double   d = (Double.parseDouble("23.45");
```

次にプログラム中で入力を得るためには、TextArea、TextFieldクラスを用いました。

```
TextField  tf = new TextField( 10 );  // initメソッドにてコンポーネントを
add( tf );                                    //  Applet上に配置する
String   line  = tf.getText();                //  actionPerformedメソッドにて
```

文字列を数値に変換する方法と、この２つの入力を得る方法のどちらかとを組み合わせれば、ユーザが入力した整数や実数をプログラム中に取り込むことができるようになります。実際のプログラムで、どのように行なっているのかみていきましょう。

★数値などの入力を受け取るアプレット

２つのテキストフィールドを用意します。ここに、２つの整数を入力します。そして、ボタンを１つ用意します。ボタンが押されたら、いくつかの演算結果をテキストエリアに表示するアプレットのプログラムになっています。initメソッドでは、これらのコンポーネントを配置しています。ボタンが押されたら実行されるactionPerformedメソッドでは、２つのテキストフィールドに入力された文字列を得て、それぞれをすぐに整数に変換して変数に代入しています。そして、この２つの変数に対して、いろいろな演算を行なった結果を再度文字列に変換して、テキストエリアに表示させています。

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

public class NumberOperation extends Applet implements  ActionListener {
        TextField       operand1, operand2;       // 入力値を受け取るためのフィールド
        TextArea        result;                   // 結果を表示するためのエリア
        Button   kick;                            // 計算を実行させるためのボタン

        public void init( ) {
                operand1 = new TextField( 10 );
                operand2 = new TextField( 10 );
                result = new TextArea( 7, 20 );
                kick = new Button( "Operation" );
                kick.addActionListener( this );

                add( operand1 ); add( operand2 );
                 add( kick ); add( result );
        }

        public void actionPerformed( ActionEvent e ) {
                int op1 = Integer.parseInt( operand1.getText() );
                int op2 = Integer.parseInt( operand2.getText() );
                String   lsep  = System.getProperty( "line.separtor" );
                result.setText(      "sum " + (op1 + op2) +  lsep +
                                     "difference " + (op1 - op2) +  lsep +
                                     "product " + (op1 * op2) +  lsep +
                                     "quotient " + (op1 / op2) +  lsep +
                                     "remainder " + (op1 % op2) +  lsep +
                                     "greater " + ((op1 > op2) ? op1 : op2) );
                repaint( );
        }
}
```

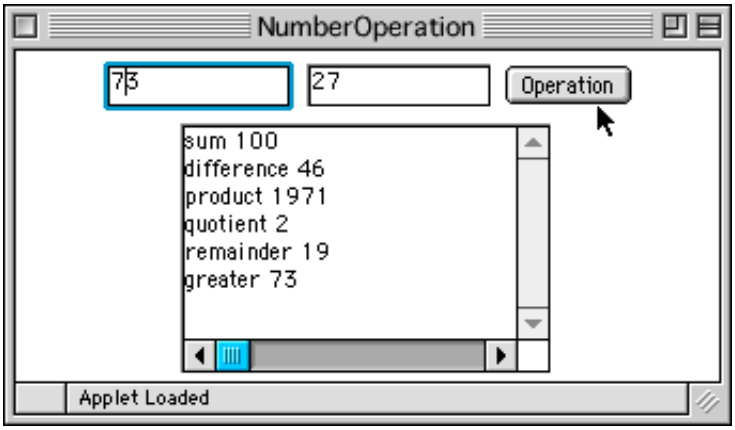

図17-4　NumberOperationアプレットの実行例

１７－３－６．文字列のフォーマット

java.textパッケージには、文字列を解析したり、フォーマットして表示したりする便利なクラスが用意されています。そのうち、ここでは、次の２つのクラスを紹介します。

数値用	NumberFormat
日付・時間用	DateFormat

★NumberFormatを利用して、数値の桁数を制御する

まず、数値用のNumberFormatクラスですが、これには、次のような整数・実数の表示桁数を調べたり、設定したりするオブジェクトを用意するメソッドが、クラスメソッド（クラス直轄のメソッド）として用意されています。

NumberFormat.getInstance( )	標準のオブジェクトが取り出される、あるいは
NumberFormat.getNumberInstance( )	

このメソッドで用意されるオブジェクトには、次のようなメソッドが用意されています。

setMinimumIntegerDigits(桁数)	整数部分の最小桁数を設定する
setMaximumIntegerDigits(桁数)	整数部分の最大桁数を設定する
setMinimumFractionDigits(桁数)	小数部分の最小桁数を設定する
setMaximumFractionDigits(桁数)	小数部分の最大桁数を設定する
setGroupingUsed(論理値)	３桁ごとに、カンマをいれるかどうか
format(値)	値（整数かlongの整数、あるいは倍精度実数）を指定されたフォーマットで文字列に直す

例えば、以下のように使います。整数部の最小表示桁数が7になっていますので、足りない部分には0が表示されます。

```
import java.text.*;

NumberFormat nf = NumberFormat.getNumberInstance( );
nf.setMaximumFractionDigits( 3 );
nf.setMinimumIntegerDigits( 7 );
nf.setGroupingUsed( false );
String message = nf.format( 12.3458372 );
```

★地域ごとのフォーマット
日付や、貨幣などについては、地域ごとに表示の仕方が異なります。そこで、Javaでは、java.utilパッケージに中にあるLocale（地域背景）という名前のクラスを使って、地域の情報を持つことができます。ここでは、Localeにクラスに標準的に用意されているいくつかの言語・地域を列挙したいと思います。

Locale.CANADA	Locale.CANADA_FRENCH	Locale.CHINA
Locale.CHINESE	Locale.ENGLISH	Locale.FRANCE
Locale.FRENCH	Locale.GERMAN	Locale.GERMANY
Locale.ITALIAN	Locale.ITALY	Locale.JAPAN
Locale.JAPANESE	Locale.KOREA	Locale.KOREAN
Locale.PRC	Locale.SIMPLIFIED_CHINESE	Locale.TAIWAN
	Locale.TRADITIONAL_CHINESE	
Locale.UK	Locale.US	

また、DateFormatクラス自体には、どの程度の長さで表示をするかのための変数が用意されています。

DateFormat.DEFAULT	
DateFormat.FULL	DateFormat.LONG
DateFormat.MEDIUM	DateFormat.SHORT

これらのLocaleクラスやDateFormatの値を使って、日付や時間の表示の仕方を地域ごとに設定することができます。

```
import   java.text.*;
import   java.util.*;

DateFormat   df = DateFormat.getDateInstance( DateFormat.DEFAULT, Locale.UK );
DateFormat   tf = DateFormat.getTimeInstance(  DateFormat.LONG,  Locale.JAPAN );
String   today = df.format(  new  Date( ) );
String   now  = tf.format(  new  Date( ) );
```

なお、日付や時間の桁数なども指定できるSimpleDateFormatクラスも用意されています。

１７－４．課題

17-1.
TextAreaを使って、すべてのUnicodeについて、表示するようなアプレットを作りなさい。クラス名は、UnicodeViewerです。16個の文字を表示するごとに、コード番号を16進数表示するようにしなさい。

17-2.
単語がいくつあるか数える例題は、単語と単語の空白の個数が必ず１つであると仮定していました。この仮定をやめて、替わりに、単語と単語の間の空白の個数は１個以上であればよいとします。正確に単語の数をカウントするようにします。そして、文字列をユーザに入力してもらい、次のような内容を１つのプログラムで表示するアプレットを作ってみなさい。クラス名は、WordCounterとします。

・入力された文字列の中にいくつ単語があるか数えて表示する。
・入力された文字列の中の大文字をすべて小文字に変えて表示する。

さらに余裕があれば、文の前後にいくつ空白が入っても（文字列自体が空白で始まっても良いし、空白で終わっても良い）、単語をちゃんと数えられるような形に改善してみなさい。

ヒント：空白かどうかチェックしているif文をwhile文に替えます。空白でなくなった段階で*wordcount*の値を１つ増やします。文の前後の空白を取るためには、trimを使います。

17-3.
１行文字列をユーザに入力してもらい、それが西暦の年を示す整数であると仮定します。その西暦に応じて、元号と年を表示するようなアプレットを作成しなさい。以下の対応表を参考に明治以降の年に対して、元号とその元号内での年を表示するようにします。クラス名は、Nengoとします

明治元年	1868
大正元年	1912
昭和元年	1926
平成元年	1989

ヒント：西暦入力用のテキストフィールドと、ボタンと、元号と年を表示するためのフィールドの３つのコンポーネントを用意しなさい。