# 情報処理概論
# (Basic Theory of Information Processing)

# 第３回：変数・演算

---

## 概要

- 変数，文字リテラル
- 文と式
- 演算子 (数値演算子，代入演算子，インクリメント・デクリメント演算子，ビット演算子，文字列演算子)

## 5.1 変数 (p.26)

- プログラムの実行中に，後で必要となるデータを変数に格納することができる．
- そして，変数からデータを取りだすことができる．

```
i = 1;      変数iに3を格納する．(変数iに3を代入すると言う．)
j = i + 4;  変数iから値を取り出し4を加算し，変数jに格納する．
```

**変数の種類** (今はよく分からなくても良い。)

**ローカル変数：**

- メソッドの中で宣言する．自身のメソッドの中だけで参照できる。(クラスの節)

**仮引数：**

- メソッドに渡れた値が格納されている。メソッドは，上位のプログラムから仮引数を通じて値をもらい，処理を進める。(メソッドの節)

**フィールド：**

- クラス宣言の中で宣言する．
- あるオブジェクトのフィールドは，そのオブジェクトのメソッドからは，自由に参照(データを読み書きする)できる．
- クラス型変数(オブジェクト参照型変数，オブジェクトを表している変数)を通して，そのオブジェクトのフィールドを参照できる。ただし，可視性の制限を受ける．

### 5.1.1 識別子

#### 識別子とは

- 変数，メソッド，クラスを区別するための名前 **(クラス名は変数ではない)**.

#### 識別子の決まり

- 使うことができる文字は，半角で英字，数字，_，$($は推奨されない).
- 大文字，小文字は区別される.
- 数字で始まってはいけない.
- 長さに制限は無い(実用上はどんなに長くても良い).
- **キーワード**と`true`, `false`, `null`は使ってはいけない.

#### キーワード

**abstract：** 抽象クラス，抽象メソッドの宣言に使う．
**boolean：** 変数の型が0 or 1のバイナリーであること示す．
**break：** ループの中から抜け出す．
**byte：** 変数の型がバイトであること示す．
**case：** 場合分けする場合に使う．(`switch`, `default`といっしょに使う)
**catch：** 例外を受け取る．
**char：** 変数の型が文字であることを示す．
**class：** クラス宣言に使う．
**const：** 変数の値が変らないことを示す．
**continue：** ループ先頭に戻る．
**default：** 場合分けの場合に，どの`case`文にも該当しない時に実行される文を示す．
**do：** ループのために使う．
**double：** 変数の型が倍精度浮動小数点であることを示す．
**else：** `if`文の条件に該当しないときに，実行される文を示す．
**extends：** 継承するクラスを指定する．
**final：** クラスを継承させないことを示す．
**finally：** 例外で最終的に実行する．
**float：** 変数の型が32 bitの浮動小数点数であることを示す．

**for：** ループのために使う.
**goto：** プログラムの実行点を移動させる．(ほとんど使わない)
**if：** 条件実行のために使う.
**implements：** インターフェースをインプリメントするために使う
**import：** 外部のクラスをインポートするときに使う.
**instanceof：** オブジェクトがあるクラスのインスタンスかどうか調べる.
**int：** 変数の型が32 bitの整数であることを示す.
**interface：** インターフェース(制約があるクラス)であること示す.
**long：** 変数の型が64 bitの整数であることを示す.
**native：** 特定のOS，CPUに依存することを示す.
**new：** オブジェクトや配列を生成するときにつかう.
**package：** パッケージを使うことを示す時に使う.
**private：** 自分のクラスだけからアクセスできる可視性を示す.
**protected：** 自分および継承したクラスからアクセスできる可視性を示す.
**public：** どのクラスからでもアクセスできる可視性を示す.
**return：** メソッドの処理から戻るときに使う.
**short：** 変数の型が16 bitの整数であることを示す.
**static：** クラス変数であることを示す.

**strictfp：** 浮動小数点をIEEE-754で処理することを示す.
**super：** 自分のスーパークラスを示す.
**switch：** 場合分けであることを示す(`case`, `default`といっしょに使う).
**synchronize：** スレッドを同期させる.
**this：** 自分自身のオブジェクトを示す変数.
**throw：** 例外を発生させる.
**throws：** 発生しそうな例外を明示する.
**transient：** シリアライズ時に保存しないことを示す.
**try：** 例外を調べるために使う.
**void：** 変数に実質的に型が無いことを示す.
**volatile：** スレッド間の内容を同期させることを示す.
**while：** 条件が成立している間文を実行する.

### 5.1.2　変数の型

- **基本データ型変数**：基本データ型の値を**格納**する変数 (p.142)
- **参照型変数**：オブジェクトや配列データを格納している位置を格納している.
  - **クラス型変数**：オブジェクト参照型変数 (p.272)
  - **配列変数**：配列の参照型変数 (p.178)

**基本データ型**

| 型 | 内容 | bit | 値の例 |
|---|---|---|---|
| boolean | 真偽値 | 1 | true, false |
| char | 文字 (unicode) | 16 | 'a', '漢', '字',... |
| byte | 符号付き整数 | 8 | − 128~127 |
| short | 符号付き整数 | 16 | − 32768~32767 |
| int | 符号付き整数 | 32 | $-2^{31} \sim 2^{31} - 1$ |
| long | 符号付き整数 | 64 | $-2^{63} \sim 2^{63} - 1$ |
| float | 浮動小数点 | 32 | IEEE-754 規格 |
| double | 浮動小数点 | 64 | IEEE-754 規格 |

- 基本データ型は上記だけ.
- これらを組み合わせてデータ構造を作る.

### 5.1.3　変数宣言

**変数宣言**：ある識別子を変数として使うと宣言すること

- その識別子が変数名となる。

**基本データ型変数宣言やクラス型変数宣言の書き方**)：

型 識別子;
型 識別子, 識別子, 識別子;

- 型には，基本データ型の名前やクラス名(クラスを示す識別子)が書かれる。
- 型と識別子の間には**ホワイトスペース(スペース，タブ，改行)**を入れる。
- 識別子が変数名を表す。
- 変数宣言は **文**である。
- **文**の終りには，; (セミコロン)が必要である。

**例：**

```
int i;
int seiseki_yamashita;
int seiseki_kawakami;
byte b3;
short si;
char c;
long  nenshuu;
double heikinn;
int x;
```

識別子i，seiseki_yamashita，seiseki_kawakami，xをint型として，b3をbyte型として，siをshort型として，cをchar型として，nenshuu，heikinnをdouble型として宣言している。

## 5.2 リテラル

- **リテラル**とはプログラム中に書かれる具体的な数や文字のこと

### 5.2.1 整数型のリテラル

| 型式 | 例 | 型 |
|---|---|---|
| 通常の符号付き整数 | 10, -1000, 521 | 10進数，int型 |
| 符号付き整数直後にLを付加 | 10L, -1000L, 521L | 10進数，long型 |
| 符号の後に0を付加 | 010, -01000, 0521 | 8進数，int型 |
| 符号の後に0を付加し，数の直後にLを付加 (整数部の各桁の値は，0〜7) | 010L, -01000L, 0521L | 8進数，long型 |
| 符号の後に0xを付加 | 0x1a, -0x1A, 0x5c2 | 16進数，int型 |
| 符号の後に0xを付加し，数の直後にLを付加 (整数部の各桁の値は，0〜9，A〜F) | 0x1aL, -0x1AL, 0x5c2L | 16進数，long型 |

- 16進表記で使われるアルファベットは，大文字(A〜F)でも小文字(a〜f)のどちらでも良い.
- 16進表記の0xも，0Xでもよい.

### 5.2.2 浮動少数点型のリテラル

| 型式 | 例 | 型 |
|---|---|---|
| 小数 | `0.123`, `-3.14` | `double`型 |
| 指数付き小数 | `3.0e-3`, `-.074e-6` | `double`型 |
| 小数にFを付加 | `0.123F`, `-3.14F` | `float`型 |
| 指数付き小数にFを付加 | `3.0e-3F`, `-.074e-6F` | `float`型 |

### 5.2.3 bool型のリテラル

| 表記 | 意味 |
|---|---|
| `true` | 真 |
| `false` | 偽 |

### 5.2.4 文字型のリテラル (p.82)

```
char c1 = 'A';
char c2 = '\u5c71'; (ユニコードの「山」)
char c3 = '山';
```

エスケープシークエンス (Javaで意味がある文字を表すときに使う) の例：

| | |
|---|---|
| \n | 改行 |
| \' | シングルクォーテーション |
| \\ | バックスラッシュ(または円マーク) |

Javaの文字は，ユニコードが使われる．従って，1文字はすべて16bitである．

**参考**：漢字コード

- ISO 8859 : ラテン語系の文字コード (8bit)
- ISO 2022 : 多種の文字を切り替えて使う (16 bit)
  ISO-2022-JP(JISコード), Shift JIS, EUC(UNIXで使う)
- ISO 10646 (Unicode) : 世界のすべての文字を表わす (16 bit)

## 5.3 整数型変数の内部表現(2の補数表現)

- **整数型：** `byte`, `short`, `int`, `long`
- 整数型変数では**原則的には2進数によって整数を表す.**
- しかしながら，マイナスの数を表すための工夫が必要.
- 最上位ビットを**符号ビット**とする.

$$\text{整数変数} \begin{cases} \geq 0 \ (\text{符号ビット} = 0) \\ < 0 \ (\text{符号ビット} = 1) \end{cases}$$

従って，数字を表わす部分のビット数は以下のようになる.

| `byte` | `short` | `int` | `long` |
|---|---|---|---|
| 7 bit | 15 bit | 31 bit | 63 bit |

- 正の数に関しては，整数を2進数にしたものをメモリーに格納する.
- 負の数を表す場合は，数の部分を変換する．$\Rightarrow$ **2の補数表現**
- 2の補数表現を使うことによって，正の数を計算するときと，負の数を計算するときでアルゴリズムを変える必要がなくなる.

## 2の補数表現

> 2の補数表現とは，数を表す部分の各ビットを反転(0を1に，1を0に入れ換える)して，1を加算したもの．

参考：1の補数表現とは，数を表す部分の各ビットを反転したもの．

**例：**

- byte型(8 bit)を考える．
- $+37$は00100101となる．
- $-37$は，まず0100101(37)の部分の2の補数を求め，符号ビットを付加する．
  1. ビット反転して1011010になる．
  2. これに1を加算した1011011が2の補数になる．
  3. 符号bitを付けた11011011がbyte型の$-37$である．
- 加算の場合は，例えば，$57 + (-37)$の場合でも，2進数として加算して，9bit目に繰り上がりがある場合にはそれを無視すればよい．

$$\begin{aligned} 00111001(57) + 11011011(-37) &= 100010100 \\ &\Rightarrow 00010100(20) \quad (9\text{ビット目を無視}) \end{aligned}$$

- 減算に場合は，引く数の符号ビットを反転し，数の部分を2の補数に直して加算すればよい．

$$\begin{aligned} 00111001(57) - 00100101(37) &= 00111001(57) + (11011010 + 1) \\ &= 00111001(57) + 11011011(-37) \\ &= 100010100 \\ &= 00010100(20) \quad (9\text{ビット目を無視}) \end{aligned}$$

- この例を使って，2の補数を使えば減算が加算に変わることを説明する．

$$11111111 + 1 = 100000000$$

であるから，

$$\begin{aligned} 00111001 - 00100101 &= 00111001 + ((11111111 - 00100101) + 1) - 100000000 \\ &= 00111001 + 11011011 - 100000000 \end{aligned}$$

となり，2の補数と加算の結果で，9 bit目を無視すれば良いことがわかる．
**ここで，11111111から引くことがビット反転に相当している．**

- **2の補数の2の補数を計算するともとの数に戻る．**37の2の補数1011011の2の補数は，0100100+1=0100101となり，37に戻る．
- 引く数が負の数の場合でも，符号ビットを反転し，数部分の2の補数の変換を行ない加算する．

- 10進数で，10の補数と9の補数を考えてみよう！
  10の補数を使えば加算で減算が実現できる。

$$523 - 454 = 523 + (999 - 454 + 1) - 1000 = 523 + 546 - 1000 = 69$$

  において，$999 - 454 + 1 = 546$が454の10の補数になる。

## 5.4 IEEE-754

- 浮動小数点の数を表わす標準的フォーマット
- 単精度(32 bit のデータ)と倍精度(64 bit のデータ)がある.

### 単精度

- 32 bit のデータを，1 bit の符号部，8 bit の指数部，23 bit の仮数部に分ける.

| | | |
|---|---|---|
| $s$ | $y_7y_6y_5y_4y_3y_2y_1y_0$ | $x_{22}x_{21}x_{20}x_{19}x_{18}x_{17}x_{16}x_{15}x_{14}\cdots\cdots\cdots\cdots\cdots x_5x_4x_3x_2x_1x_0$ |

- $y$で指数部が表わす整数(0〜255)とする.
- $x$で仮数部が表わす整数(0〜8288607)とする.
- 単精度のIEEE-754の浮動小数点の数が表わすものは以下の通り.

| 指数部$y$ | 仮数部$x$ | 表わす数 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | $\neq 0$ | 非正規化数：$(-1)^s \times x_{22}.x_{21}x_{20}x_{19}\cdots\cdots\cdots\cdots\cdots x_2x_1x_0 \times 2^{-127}$ |
| 1〜254 | 任意 | 正規化数：$(-1)^s 1.x_{22}x_{21}x_{20}x_{19}\cdots\cdots\cdots\cdots\cdots x_2x_1x_0 \times 2^{y-127}$ |
| 255 | 0 | 無限大(s=0ならば正，s=1ならば負) |
| 255 | $\neq 0$ | 非数(NaN, Not a Number，0を0で割ったなどの場合に生じる) |

最大の正数：$1.11\cdots111 \times 2^{254-127} = (2 - 1/2^{23}) \times 2^{127}$
最小の正数：$0.00\cdots01 \times 2^{0-127} = 2^{-149}$ (非正規化数)

**例：**

| | | |
|---|---|---|
| 0 | 10000000 | 10000000000000000000000 |

の場合は，以下のようになる．

$$(-1)^0 \times 1.10000000000000000000000 \times 2^{128-127} = 3.0$$

## 倍精度

- 基本的には単精度と同様である．
- 64 bit のデータを，1 bit の符号部，11 bit の指数部 (1023 を引く)，52 bit の仮数部に分ける．

## 丸め

- 演算器における計算の途中では，仮数部を2 bit増やし計算精度を高める．
- もとのbit長にもどすために，丸めを行なう．次の方法が用意されている．
  - 切り上げ ($+\infty$へ)
  - 切り捨て ($-\infty$へ)
  - 最も近い偶数への丸め (四捨五入の改良) 小数点1桁めを偶数へ丸める例：
    $0.0 \to 0$，$0.1 \to 0$，$1.0 \to 1$，$1.1 \to 10$，
    $10.0 \to 10$，$10.1 \to 10$，$11.0 \to 11$，$11.1 \to 100$

# 6　演算 (p.86)

## 6.1　文 (statement)

- 文はプログラムの処理の単位
- ; で文が終了する。
- 例：
  - 空文：; だけ
  - 式文：`x = y + z;` (式からなる文)
  - if 文：`if (x == y) System.out.println("x と y は等しい。");`
  - while 文：`while (x < y) System.out.println("x は y 未満の間");`

## 6.2　式 (expression)

- 式は演算を記述する。
- 式は**演算子** (operator) と**オペランド** (operand) によって構成される.
- オペランドとなるものは，変数とリテラルと式である。
- 演算子は演算の種類を表わすもの。あかじめ決まっている。　`+`，`-`，`*`，`/`など
- プログラムを実行すると式には値が決まる。プログラムの実行時に式の値を求めることを，**「式を評価 (evaluate) する」**という。

## 6.3　演算子

### 単項演算子と2項演算子

- **単項演算子** (例：`-a`，`a++`)：
  - オペランドは1つだけ.
  - オペランドは演算子の前あるいは後に置かれる.
- ２項演算子 (例：`a + b`)
  - **オペランド**は2つだけ.
  - オペランドはオ演算子の前後に置かれる.

### 評価の順番

- 優先順位
  - `a + b * c` ⇔ `a + (b * c)`
- 評価する方向 (右から左，左から右)
  - `a + b + c` ⇔ `(a + b) + c`
  - `a = b = c` ⇔ `a = (b = c)`

## 演算子の種類

- **数値**演算子
- **代入**演算子 (代入演算子の短縮記法)
- **インクリメント・デクリメント**演算子
- **比較**演算子
- **論理**演算子
- **ビット**演算子
- **文字列**演算子

**注意：** 数値演算子，代入演算子，インクリメント・デクリメント演算子をあわせて**算術**演算子とも呼ぶ

## 数値演算子

| 演算子 | 意味 | 例 |
|---|---|---|
| + | 加算 | a + b ：aとbを加算したものを値に持つ |
| - | 減算 | a - b ：aからbを減算したものを値に持つ |
| * | 乗算 | a * b ：aとbを乗算したものを値に持つ |
| / | 除算 | a / b ：aをbで除算したものを値に持つ<br>(整数演算での注意：-18 / 5は，-3) |
| % | 余り計算 | a % b ：aをbで割った余りを値に持つ<br>(整数演算での注意：-17 % 5は，-2) |

## 代入演算子

- 先の数値演算子だけでは，計算は行なうが結果が残らない．
- 従って，通常は数値演算の後に代入・比較演算が伴う．

| 演算子 | 意味 | 例 |
|---|---|---|
| = | **代入演算子** | a = b ：bの値をaに代入する |
| += | 加算後代入 | a += b ：aとbを加算したものをaに代入する |
| -= | 減算後代入 | a -= b ：aからbを減算したものをaに代入する |
| *= | 乗算後代入 | a *= b ：aとbを乗算したものをaに代入する |
| /= | 除算後代入 | a /= b ：aをbで除算したものをaに代入する |
| %= | 余り計算後代入 | a %= b ：aをbで割った余りをaに代入する |

## インクリメント・デクリメント演算子

| 演算子 | 意味 |
|---|---|
| | 例 |
| ++ | **インクリメント**(1を加算) |
| | ++a　：aに1を加算してからaの値を参照する.<br>a++　：aの値を参照してからaに1を加算する. |
| -- | **デクリメント**(1を減算) |
| | --a　：aから1を減算してからaの値を参照する.<br>a--　：aの値を参照してからaから1を減算する. |

## 複合した例

```
kazu = num + abc;
c = a - b;
d = a * b;
wad = a / b;
amari = a % b;

iroiro = a * (b + c /(d - 3));

junnban = a * b + c;

a += b + c;
     (a = a + (b + c); と同じ)
a %= b + c;
     (a = a % (b + c); と同じ)

a++;   (a = a + 1; と同じ)
++a;   (a = a + 1; と同じ)
```

## ++a と a++ の違い

```
b = a++; (b = a;
           a = a + 1; と同じ)
b = ++a; (a = a + 1;
           b = a; と同じ)
```

いまaが3のとき,

| 演算 | 実行結果 |
|---|---|
| b = a++; | aは4<br>bは3 |
| b = ++a; | aは4<br>bは4 |

となる.

## 6.4 型変換

- 演算子の左右のオペランドで型が異るとき**型変換**が行なわれる.
- 整数型，浮動小数点型では，表わすことができる数の大きさについて次のような決まりがある.

```
byte < short < int < long < float < double
       char < int < long < float < double
```

- 数値演算子の場合は，**下位の型の値が上位の型の値に変換されてから演算**される.
- 代入演算子に関しては,
  - 左辺(lhs, left hand side)が，右辺(rhs, right hand side)よりも上位の場合，**右辺の値を左辺の型に変換してから代入する.**
  - そうでない場合は**コンパイル時にエラー**となる.
    ⇒ 明示的に型変換を示すために**キャスト**する.

### キャスト

- 型変換したいものの直前に，(型)を付加する.

```
int i;
double d;
i = (int) d;            ⇐   dがint型に変換されてから代入される.
d = (double) i + 3;     ⇐   iがdouble型に変換され，double型の加算が行なわれる.(3もdouble型になる.)
```

## 確認するプログラム

```
public class Cast {
 public static void main(String args[]) {
  int i = 1, j = 11;
  double a = 4.0, b = 14.0;
  b = i;
  //j = a;
  j = (int) a;
  System.out.println("i = " + i + " j = " + j + " a = " + a + " b = " + b);

  i = j = 3;
  a = i / 2 + 3;
  b = (double) j / 2 + 3;
  System.out.println("i = " + i + " j = " + j + " a = " + a + " b = " + b);
 }
}
```

## 6.5 比較演算子

条件判断の結果の値の型は`boolean` (`true`, `false`)である.

### 6.5.1 条件演算子

| | |
|---|---|
| `x == y` | xとyが**等しい**ときにtrue，そうでなければfalse |
| `x != y` | xとyが**等しくない**ときにtrue，そうでなければfalse |
| `x > y` | xがy**より大きい**ときtrue，そうでなければfalse |
| `x < y` | xがy**より小さい**ときtrue，そうでなければfalse |
| `x >= y` | xがy**以上**のときtrue，そうでなければfalse |
| `x <= y` | xがy**以下**のときtrue，そうでなければfalse |

### 6.5.2 論理演算子

| | |
|---|---|
| `a & b` | aとbの論理積 |
| `a \| b` | aとbの論理和 |
| `a ^ b` | aとbのEXOR |
| `! a` | aの否定 |
| `a && b` | aとbの論理積 |
| `a \|\| b` | aとbの論理和 |

| `a` | `b` | `a & b`<br>`a && b` | `a \| b`<br>`a \|\| b` | `a ^ b` |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

| `a` | `! a` |
|---|---|
| 0 | 1 |
| 1 | 0 |

0 : `false`

1 : `true`

## &と&&の違い

- a && bは，aがfalseならばbを評価せずにfalseを出力する．
  aがtrueならばbも評価する
- a & bは，aとbを評価してから，論理積を計算する．
- a || bは，aがtrueならばbを評価せずにtrueを出力する．
  aがfalseならばbも評価する．
- a | bは，aとbを評価してから，論理和を計算する．

## 条件演算子を組み合わせた例

| 例 | 意味 |
|---|---|
| `x >= y & x <= z` | $y \leq x \leq z$ならばtrue. |
| `x < y \| x > z` | $x < y$または$x > z$ならばtrue. |
| `!(x >= y & x <= z)` | `x < y \| x > z`と同じ．<br>(ド・モルガンの法則) |

```
public class AndAnd {
 public static void main(String[] args) {
  if (check("p11", 1, 1) && check("p12", 1, 1)) pOk("p1"); else pNg("p1");
  if (check("p21", 1, 1) && check("p22", 1, 2)) pOk("p2"); else pNg("p2");
  if (check("p31", 1, 2) && check("p32", 1, 1)) pOk("p3"); else pNg("p3");
  if (check("p41", 1, 2) && check("p42", 1, 2)) pOk("p4"); else pNg("p4");

  if (check("q11", 1, 1) & check("q12", 1, 1))  pOk("q1"); else pNg("q1");
  if (check("q21", 1, 1) & check("q22", 1, 2))  pOk("q2"); else pNg("q2");
  if (check("q31", 1, 2) & check("q32", 1, 1))  pOk("q3"); else pNg("q3");
  if (check("q41", 1, 2) & check("q42", 1, 2))  pOk("q4"); else pNg("q4");
 }
 // 表示するためのメソッドある
}
```

結果：

```
 p11 = true  p12 = true  p1 Ok
 p21 = true  p22 = false  p2 Ng
 p31 = false  p3 Ng
 p41 = false  p4 Ng
```

```
 q11 = true  q12 = true  q1 Ok
 q21 = true  q22 = false  q2 Ng
 q31 = false  q32 = true  q3 Ng
 q41 = false  q42 = false  q4 Ng
```

表示するためのメソッド：

```
// iとjが等しければ，trueを表示し，戻り値もtrueとなるメソッド
static boolean check(String str, int i, int j) {
  System.out.print(str + " = " + (i == j) + "  ");
  return (i == j);
}
// strと Okをつなげて表示するメソッド
static void pOk(String str) {
  System.out.println(str + " Ok");
}
// strと Ngをつなげて表示するメソッド
static void pNg(String str) {
  System.out.println(str + " Ng");
}
```

## 6.6 ビット演算子

- **整数型の変数をbit列と考えて，同じ桁どうしでビット演算を行なう．**
  論理としては，0がfalseで，1がtrueと考える．

| 演算子 | 意味 |
|---|---|
| & | ビットごとのAND |
| \| | ビットごとのOR |
| ^ | ビットごとのEXOR |
| << | x<<yは，yビットだけxを左シフト |
| >> | x>>yは，yビットだけxを(算術)右シフト |
| >>> | x>>>yは，yビットだけxを(論理)右シフト |
| ~ | ビットごとのNOT |
| &= | ビットごとのANDを計算して代入 |
| \|= | ビットごとのORを計算して代入 |
| ^= | ビットごとのEXORを計算して代入 |
| <<= | x<<=yは，yビットだけxを左シフトしてxに代入 |
| >>= | x>>=yは，yビットだけxを(算術)右シフトxに代入 |
| >>>= | x>>>=yは，yビットだけxを(論理)右シフトxに代入 |

## 6.7 文字列演算子

- **文字列**とは，文字を列べたものである．
- 人間とインターフェースするために非常に重要．
- Javaでは，文字列は基本データ型ではなく**オブジェクト**である．
- 文字列のクラスは，Stringである．
- 文字列のリテラルは，ダブルクォーテーションで挟んで示す．
- + 演算子で文字列を連結することができる．

  `"誰でもわかる" + "やさしい言語" + "Java"`

  は，

  `"誰でもわかるやさしい言語Java"`

  と同じ．
- + 演算子のオペランドがString型の変数の場合も，変数に格納されている文字列が連結される．
- 整数型・浮動小数点型などのものと + で演算すると，**整数型・浮動小数点型の値が文字列に変換されてから連結される．**
  整数型変数iに3が代入されているとき，`"x = " + i +"."`は，`"x = 3."`と同じ．

## 6.8　演算子の優先順位

**さて，**`h = i + j * k`**が演算される順番は？**

- 演算子に優先順位が決まっている (次ページの表を参照).
- **優先順位が高い方から演算される．**
- 同じ優先順位の演算子が並んでいる場合，右から左，または，左から右に演算する方向が決まっていて，その方向から演算する (**オペランドが隣り合うことはない**).

**式を評価する順番の説明 (概略)　(詳細は，構文解析器による)**

- 式1が ( 式2 ) という形ならば，括弧の中の式2を評価 (計算) したものを式1の評価とする。
- 括弧の中に含まれていない，その式の中で最も低い優先順位の演算子を探す。
- その最低の優先順位の演算子の中で，定めらたれ方向で最後の位置の演算子を選び出す。
- その演算子が，単項演算子の場合は，オペランドとなる片側を1つの式として評価し，その結果をオペランドして演算する。
- その演算子が**左から右**に評価する2項演算子の場合は，**左側**にあるものを1つの式として評価し，次に**右側**にあるものをもう1つの式として評価し，それらの結果をオペランドとして演算する。演算する方向が逆の場合も同様である。

| 優先 | 演算子 | 例 | 方向 |
|---|---|---|---|
| 高 | . [] (引数の並び) ++ -- | i++, i-- | 左から右 |
| | + - ++ -- ! ~ | -i, ++i, --i | 右から左 |
| | (型) new | (int) d, new String() | 右から左 |
| | * / % | a * b | 左から右 |
| | + - | a + b | 左から右 |
| | << >> >>> | x << 3 | 左から右 |
| | < > <= >= instanceof | i >= j | 左から右 |
| | == != | x == y | 左から右 |
| | & | x & y | 左から右 |
| | ^ | x ^ y | 左から右 |
| | \| | x \| y | 左から右 |
| | && | x && y | 左から右 |
| | \|\| | x \|\| y | 左から右 |
| | ? : | x == y ? a : b | 左から右 |
| | = += -= *= /= %= | x = y, x += 2 | 右から左 |
| | &= = <<= >>= >>>= | | |
| 低 | , | | 左から右 |

```
a = b + c + d;
a = b + c * d;
a = b + c + d * e;
a = b * c + d + e * f;
a = b = c * d == e * f && g + h >= i + --j;
```

は，以下の順番で評価される．

```
(a = ((b + c) + d));
   3     1    2

(a = (b + (c * d)));
   3    2    1

(a = ((b + c) + (d * e))));
   4     1    3    2

(a = (((b * c) + d) + (e * f)));
   5      1    2    4    3

(a = (b = (((c * d) == (e * f)) && ((g + h) >= ((i + (--j)))))));
   10   9      1     3    2      8     4     7     6   5
```

```
a = b - + - + - c;
c = 4 - - - i;
d = i + -- j;
```

は，以下の順番で評価される．

```
(a = (b - (+ (- (+ (- c))))) ;
   6    5  4  3  2  1

(c = (4 - (- (- i))));
   4    3  2  1

(d = (i + (-- j)));
   3    2   1
```

## 6.9　今日の最後に

- 基本データ型は全ての基礎
- 数の計算機での内部表現を知っておくことは，プログラムのバグ (間違い) を探すときに便利
  (結果から原因を推測)
- 偶数への丸めは 10 進数で考えてみると分りやすい **(四捨五入は不平等)**
- 演算を組み合わせて様々な機能を実現できる。
- ビット演算は，ハードウエアに関係あるような場合しか用いられないが，それ以外の演算子は頻繁に使うので，しっかりと覚えるようにする。