

プログラミング

Javaプログラミング
変数宣言, 入出力, 乱数

堀田 敬介

変数宣言

```
int number;  
boolean flag;
```

```
number = 3 + 5;  
flag = true;
```

整数型 (int) の変数宣言

論理値型 (boolean) の変数宣言

変数 number に計算結果を代入

変数 flag に論理値を代入

変数型	定義型	変数範囲
論理値型	boolean	true, false
整数型	byte	-128 ~ +127
	short	-32,768 ~ +32,767
	int	-2,147,483,648 ~ 2,147,483,647
	long	-9,223,372,036,854,775,808 ~ +9,223,372,036,854,775,807
浮動小数点型	float	±3.40282347E+38 ~ ±1.40239846E-45
	double	±1.79769313486231507E+378 ~ ±4.94065645841246544E-324
文字列型	String	文字列 (スペース・改行等で区切り)
	String	文字列 (1行)

キーボードからの数値・文字列の読込

Scannerクラス

```
import java.util.Scanner;
```

java.utilパッケージのScannerクラスを呼び出し

```
Scanner stdIn = new Scanner(System.in);
```

宣言, System.in=標準入力ストリーム

```
System.out.print("値を入力してね →");
```

```
int num = stdIn.nextInt();
```

キーボードからの値を「整数型で」読み込み

	メソッド	読込型	読み込める値とその範囲
論理値	nextBoolean()	boolean	true, false
整数	nextByte()	byte	-128 ~ +127
	nextShort()	short	-32,768 ~ +32,767
	nextInt()	int	-2,147,483,648 ~ 2,147,483,647
	nextLong()	long	-9,223,372,036,854,775,808 ~ +9,223,372,036,854,775,807
浮動小数点	nextFloat()	float	±3.40282347E+38 ~ ±1.40239846E-45
	nextDouble()	double	±1.79769313486231507E+378 ~ ±4.94065645841246544E-324
文字列	next()	String	文字列(スペース・改行等で区切り)
	nextLine()	String	文字列(1行)

一様疑似乱数生成

Randomクラス

```
import java.util.Random;
```

```
Random rnd = new Random();  
int dat = rnd.nextInt(10);
```

java.utilパッケージのRandomクラスを呼び出し

宣言, Random(n)とすると種nを使うことになる
整数0,1,2,...,9の一様疑似乱数(Int型)を生成

	メソッド	乱数型	生成される乱数の値
論理値	nextBoolean()	boolean	true か false のどちらか1つ
整数	nextInt()	int	-2,147,483,648 ~ 2,147,483,647 から1つ
	nextInt(n)	int	0,1,2,...,n-1 から1つ
	nextLong()	long	-9,223,372,036,854,775,808~+9,223,372,036,854,775,807 から1つ
浮動小数点	nextFloat()	float	0.0以上, 1.0未満 のfloat型1つ
	nextDouble()	double	0.0以上, 1.0未満 のdouble型1つ

条件分岐1: if文

構文の基本型)

注) 論理式: 値(結果)が論理値 true か false を返す式

```
if ( 論理式 ) {  
    ...  
}
```



論理式の値が true なら
{ ... } を実行
論理式の値が false なら
何もしない
({ ... } の次の行の命令へ)

```
if ( 論理式 ) {  
    ...  
} else {  
    ...  
}
```



論理式の値が true なら
最初の { ... } を実行し,
論理式の値が false なら
else の次の { ... } を実行

```
if ( 論理式1 ) {  
    ...  
} else if ( 論理式2 ) {  
    ...  
}
```



論理式1の値が true なら
最初の { ... } を実行し,
論理式1の値が false なら
else if の論理式2を評価,
その値が true なら
else if の次の { ... } を実行
論理式2の値が false なら
何もしない

使用例)

```
if ( month == 2 ) {  
    day == day + 1;  
}
```

比較演算子

注) **関係演算子** (二項演算子の種類の一つ)

「==」: 両辺が等しいなら true, o.w. false を返す

「!=」: 両辺が異なるなら true, o.w. false を返す

「>=」: 左辺が右辺以上なら true, o.w. false を返す

「<=」

「>」

「<」

注) **論理演算子** (二項演算子の種類の一つ)

「&&」:

「||」:

注) o.w. = otherwise
「そうでなければ」

```
if ( year == 2011 ) {  
    ...  
}
```

繰り返し1: for文

構文例

注) $i++$ の「 $++$ 」はインクリメント演算子. 「 $i = i + 1$ 」と同じ意味
注) $i--$ の「 $--$ 」はデクリメント演算子. 「 $i = i - 1$ 」と同じ意味

```
for ( i=0; i<=5; i++ ) {  
    ...  
}
```



インデックス変数 i を0から5まで動かして
繰り返し

$i = 0$... 初期化
 $i <= 5$... 終了判定式
 $i++$... 更新式 ($i = i + 1$)

- ① $i = 0$ とする. $i <= 5$ を満たすので1回目
{ }内実行, $i++$ として次へ
- ② $i = 1$ は $i <= 5$ を満たすので2回目 { }内実
行, $i++$ として次へ

.....

<以下繰り返し>

```
for ( i=5; i>=0; i-- ) {  
    ...  
}
```



インデックス変数 i を5から0まで動かして
繰り返し

$i = 5$... 初期化
 $i >= 0$... 終了判定式
 $i--$... 更新式 ($i = i - 1$)

- ① $i = 5$ とする. $i >= 0$ を満たすので1回目
{ }内実行, $i--$ として次へ
- ② $i = 4$ は $i >= 0$ を満たすので2回目 { }内実
行, $i--$ として次へ

.....

<以下繰り返し>

繰り返し2: while文 (繰り返し回数が決まってないときによく用いる)

構文例

```
i = 0;
while ( i <= 5 ) {
    ...
    i++;
}
```

↑

```
for ( i = 0; i <= 5; i++ ) {
    ...
}
```

と同じ (i=0~5 の6回繰り返ししたい時)

```
boolean flg = true;
while ( flg ) {
    ...
    if ( ... ) {      何らかの条件(...)を満たしたら
        flg = false;  flg を false にしループ脱出
    }
}
```

↑

繰り返し数を指定しない書き方の例
論理変数 flg を導入し、
flg が true である限り繰り返す。
何かの条件 (if文) が満たされると、flg =
false として繰り返し処理から抜ける

条件分岐2: **switch**文(分岐が多いときによく用いる)

構文例

```
int select;  
...  
switch ( select ) {  
    case 0: .....; break;  
    case 1: .....; break;  
    case 2: .....; break;  
    default: .....; break;  
}
```

↑
select の値が 0,1,2,o.w. で実行処理を変える
select = 0 の時, case 0: 内を実行
select = 1 の時, case 1: 内を実行
select = 2 の時, case 2: 内を実行
o.w. の時, default: 内を実行

注) 対応するcase を実行後, 「**break;**」が書いてあると {}の外に処理が移るが,
「**break;**」が書いてないと, 次の行のcaseを実行する

文字と文字列の操作I

.charAt(n) メソッド

```
char moji;  
String msg = "abcdefg";
```

```
moji = msg.charAt(0);  
System.out.println("moji = " + moji);
```

「文字列msg」の1文字目を取得し、「文字moji」に代入
「moji = a」と表示される

```
moji = msg.charAt(3);  
System.out.println("moji = " + moji);
```

「文字列msg」の4文字目を取得し、「文字moji」に代入
「moji = d」と表示される

```
char moji;  
Scanner stdin = new Scanner(System.in);
```

```
moji = stdin.next().charAt(0);  
System.out.println("moji = " + moji);
```

標準入力の文字列の1文字目を取得し、mojiに代入
「bunkyo」と入力したなら「moji = b」と表示される

```
moji = stdin.nextLine().charAt(2);  
System.out.println("moji = " + moji);
```

標準入力の文字列の3文字目を取得し、mojiに代入
「bunkyo」と入力したなら「moji = n」と表示される
(注: 3文字未満の文字を入力するとエラーとなる)

文字と文字列の操作②

.length()メソッド

```
String msg = "abcdefg";
```

```
System.out.println("msgの長さは" + msg.length());
```

「文字列msg」の長さを取得

```
String msg1 = "abc", msg2 = "wxyz";
```

```
if ( msg1.length() > msg2.length() ) {
```

```
    System.out.println(" msg1 の方が文字数が多い");
```

```
} else {
```

```
    System.out.println("msg2 の方が文字数が多い");
```

```
}
```

メソッド `method`

構文例

```
public static void main(String[] args) {  
    int x = 3, y = 5, z;  
    z = addxy(x, y);  
    System.out.printf("%2d + %2d = %2d", x, y, z);  
}
```

← メソッドの呼び出し

引数は2つのint型なので、int型を2つ渡して呼び出す

```
public static int addxy(int X, int Y) {  
    int Z = X + Y;  
    return (Z);  
}
```

← 戻り値がint型なので、int型をreturnで返却

```
修飾子 戻り値 メソッド名(引数) {  
  
}
```

修飾子の種類 ... public, private, protected
static, abstract, final

戻り値に使えるもの ... 変数の型 + void(戻り値がない場合)

(注: void以外は、必ずメソッド内に `return ()` を書く必要がある)