第2章

# 式と演算子

第1章では「プログラムの中には文を並べて書く」こと、
また「変数の宣言」というJavaにおける2つのルールを学びました。
本章では、さまざまな計算を行うための「式と演算子」と、
キーボードから文字を入力したり、画面に文字を出力したり、
さらに乱数を生み出すなどの「命令実行の文」を学んでいきます。

CONTENTS


2.1 計算の文
2.2 オペランド
2.3 評価のしくみ
2.4 演算子
2.5 型の変換
2.6 命令実行の文
2.7 第2章のまとめ
2.8 練習問題
2.9 練習問題の解答

# 2.1 計算の文

## 2.1.1 計算の文とは?

第 1 章で紹介があった 3 種類の文のうち、「①変数宣言の文」はもうバッチリです♪

2 つ目の文は「②計算の文」でしたよね？

そうだよ。この文は「電子計算機」であるコンピュータにとって、とても大事な文なんだ。

計算の文とは、変数や値を用いたさまざまな計算処理をコンピュータに行わせるための文です。計算処理と言っても、いわゆる四則演算だけではありません。変数に値を代入することもコンピュータにとっては計算の一種なのです。

**リスト 2-1　変数宣言の文と計算の文**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a;             ①変数宣言の文
4      int b;
5      a = 20;            ②計算の文（代入）
6      b = a + 5;         ②計算の文（足し算して代入）
7      System.out.println(a);
8      System.out.println(b);
9    }
10 }
```

**実行結果**

```
20
25
```

6行目の「b = a + 5」のようなものを式（expression）と呼びます。見た目は数学の式のようですね。

数学… あぁ、その言葉を聞くだけで寒気がします…。

Javaの式は数学より、ずっと簡単だから大丈夫だよ。

## 2.1.2 式の構成要素

式が何からできているか、分解して見てみよう。

式「b = a + 5」を分解すると、変数「a」「b」や値の「5」、そして「+」「=」の計算記号に分けることができます。Javaを含む、多くのプログラミング言語では、この「a」「b」「5」を**オペランド**（operand）、そして「+」「=」を**演算子**（operator）と呼びます。これは簡単な式の例ですが、より複雑な式であっても同じで、**すべての式はこの2つの要素だけで構成されています**。

「+」や「=」といった演算子は式に含まれるオペランドを使って計算を行います。たとえば＋演算子は「自分の左右にあるオペランドを加算する」機能を持っています（図2-1）。Javaに、どのような演算子があり、それがどのような機能を持つかについて、詳しくは2.4節で解説するとして、まずは次節でオペランドについての理解を深めていきましょう。

図2-1 式は演算子とオペランドで構成されている

# 2.2 オペランド

## 2.2.1 リテラル

オペランドって「変数や値」と考えておけばいいですか？

だいたい合っているよ。でも、より明確にするため特に重要なオペランドである「リテラル」を紹介しよう。

オペランドの中でも数字「5」や文字列「Hello World」など、ソースコードに記述されている値のことを**リテラル**（literal）と呼びます。そして、それぞれの**リテラルはデータ型を持っています**（表2-1）。そのリテラルが、どの型の情報を表すかはリテラルの表記方法で決まります。

**表2-1　代表的なリテラルの表記方法とデータ型**

| リテラルの種類 | 表記例 | 型 |
| --- | --- | --- |
| 小数点がない数字（整数） | 30 | int |
| 小数点がない数字で末尾がLまたはl（大きな整数） | 300000L | long |
| 小数点付きの数字（精度の高い小数） | 30.5 | double |
| 小数点付きの数字で末尾がFまたはf（比較的精度の低い小数） | 30.5F | float |
| true（真）またはfalse（偽） | true | boolean |
| 引用符で囲まれた文字 | '雅' | char |
| 二重引用符で囲まれた文字列 | "Java" | String |

一見、「'A'」と「"A"」は同じもののように見えるが、引用符の違いにより別のデータ型として扱われるので注意しよう。前者はchar型の**文字**「A」で、後者はString型の**文字列**「A」だ。

「1」「'1'」「"1"」は、どれも別物ですね。気をつけなきゃ。

## 整数リテラルに関する応用記法

整数リテラルの先頭に 0x を付けると 16 進数、0 を付けると 8 進数、0b を付けると 2 進数として解釈されます。たとえば「int a = 0x11; int b = 0b0011;」と書くと、変数 a には 17、変数 b には 3 が代入されます。

また、リテラル中の任意の場所にアンダースコア記号 (_) を含めることが許されています。日常生活で「2,000,000 円」などとカンマを入れるように、「long price = 2_000_000;」のように表記することで、大きな数値もわかりやすく表記することができます。

## 2.2.2 エスケープシーケンス

String 型や char 型のリテラルを記述する際に、ときどき用いられるものが**エスケープシーケンス**（escape sequence）と呼ばれる特殊な文字です。これは次のような「¥ 記号と、それに続く 1 文字」の合計 2 文字による記述方法で、その 2 文字で特殊な 1 文字を表現します。

表 2-2　代表的なエスケープシーケンス

| 表記 | 意味 |
|---|---|
| ¥" | 二重引用符記号 (") |
| ¥' | 引用符記号 (') |
| ¥¥ | 円記号 (¥) |
| ¥n | 改行 (制御文字) |

な、なにこれ？ どうしてこんなものが必要なんですか？

「引用符記号」や「金額」を画面に表示するときなどに必要になる記号だよ。

たとえば「私の好きな記号は二重引用符（"）です」という文字列を画面に表示するプログラムを考えてみましょう。単純に考えて次のリスト2-2のように表記してしまうと、コンパイルエラーになってしまいます。

**リスト2-2　エスケープシーケンスを用いていない例（エラー）**

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("私の好きな記号は二重引用符（"）です");
4   }
5 }
```

この部分だけが文字列と見なされる

Javaは**2つの二重引用符に囲まれた部分**を文字列リテラルと見なします。そのためリスト2-2の3行目では、途中までが文字列と解釈され、最後の「）です"」は文字列とは見なされません。よって、この部分がコンパイルエラーとなります。

Javaは気がきかないなぁ。途中の「"」は文字列の終わりを表す記号じゃなくて、画面に出す文字としての「"」なのに…。

このような場合、エスケープシーケンスを用いれば、「画面に出す文字としての"である」ことをJavaに対して伝えることができます（図2-2）。

先ほどのプログラムは、リスト2-3のように改良することで期待どおりに動作します。

図 2-2　文字列中に二重引用符を含めるには
エスケープシーケンスを用いる

**リスト 2-3　エスケープシーケンスを用いた例**

Main.java

```
1 public class Main {
2   public static void main(String[ ] args) {
3     System.out.println("私の好きな記号は二重引用符（¥"）です");
4   }
5 }
```

¥" によって二重引用符は文字と見なされる

このコードをコンパイルして実行すると、画面には次のように表示されます。

**実行結果**

```
私の好きな記号は二重引用符（"）です
```

金額を表示するとき、たとえば「¥1200」と表示したい場合は、「¥¥1200」というリテラル表記をすることで正しく¥マークが表示されるんだ。

## 2.2.3 リテラル以外のオペランド

式のオペランドとして利用できるのはリテラルのほかにも「変数」「定数」「命令の実行結果」などがあります。変数や定数については、すでに第 1 章で学習しましたね。また「命令の実行結果」については、2.6 節で詳しく紹介します。

オペランドの学習は、ひとまずここで切り上げ、次節では「どのような順番で式が計算されていくか」を学びましょう。

# 2.3 評価のしくみ

## 2.3.1 評価の結果

「a = 3 + ( b + c ) * 4 ;」のような複雑な式も、Java はちゃんと実行してくれるんですね。

そうだよ。
Javaが、どのような手順で式を計算しているかを説明しよう。

Java が式に従って計算処理をすることを、式の**評価**（evaluation）と呼びます。Java は 3 つの単純な原則に従いながら、式の一部から少しずつ部分的に処理してゆき、最後には式全体の計算処理が完了します。

評価の 3 つの原則って何ですか？

まず最も重要な「評価結果への置換の原則」を紹介しよう。
これは必ず理解してほしい。

### 評価結果への置換の原則

演算子は周囲のオペランドの情報を使って計算を行い、それら**オペランドを巻き込んで結果に化ける（置き換わる）**。

たとえば「1 + 5」という式の場合、+演算子はオペランド1と5を使い、それらを足した計算結果「6」に化けます。

1 + 5 → 評価 → 6

図2-3 演算子はオペランドを巻き込んで結果に「化ける」

より複雑な式「1 + 5 - 3」の場合には、段階的に評価が行われていきます。まず「1 + 5」の部分が「6」に化けて「6 - 3」という式に変形されます。それが処理された結果の「3」に化けて計算は終了です。

1 + 5 - 3 → 評価 → 6 - 3 → 評価 → 3

図2-4 「1 + 5 - 3」→「6 - 3」→「3」と変化する

## 2.3.2 優先順位

式に複数の演算子が含まれることもあると思いますが、どの部分から順に評価していくんですか？ 左の演算子から順番にかな？

いや、そうとは限らないよ。
演算子には「優先順位の原則」があるんだ。

### 優先順位の原則

式に演算子が複数ある場合は、**Javaで定められた優先順位の高い演算子から順に**評価される。

Javaには多くの演算子がありますが、それらには優先順位（15段階）が定められています（優先順位は次節で解説します）。たとえば「5番目に優先」の演算子グループに属している＋演算子よりも、「4番目に優先」のグループに属する＊演算子のほうが先に評価されます。「1 + 5 * 3」という式があれば、先に掛け算が行われるのです。もし、式の中で「1 + 5」の評価を優先したい場合は、丸カッコ「()」を使うことで評価順位を引き上げることができます。

+より＊のほうが優先順位が高い演算子なので…

()で囲んだ範囲は先に評価される

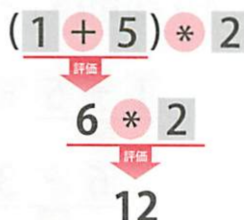



図2-5　優先度が高い演算子から評価される

すべての演算子の優先順位をがんばって覚えなきゃ！

いや、そんな必要はないよ。プログラムを書いていれば自然に覚えられるから、必要なときに調べれば十分だよ。

## 2.3.3　結合規則

もし「同じ優先順位の演算子」が2つ以上あったら、どっちが優先されるんですか？

そのルールは「結合規則の原則」で決められているんだ。

## 結合規則の原則

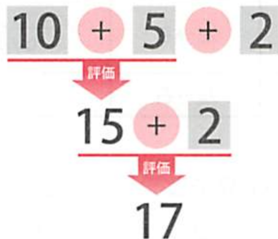
式の中に同じ優先順位グループに属する演算子が複数ある場合、演算子ごとに決められた「方向」から順に評価される。

すべての演算子には、左から評価をするか、または右から評価をするかという「方向」が結合規則として定められています。たとえば+演算子は左から右へ評価しますので「10 + 5 + 2」は次のようになります。

①「10 + 5」を評価して結果は「15」。
②「15 + 2」を評価して結果は「17」。

一方、=演算子は右から左へ評価しますので「a = b = 10」という式の場合は次のようになります。

①最も右の = 演算子に関する「b = 10」が評価され、b に 10 が代入された結果、式自体は「10」に化ける。
②次に「a = 10」が評価され、a に 10 が代入される。



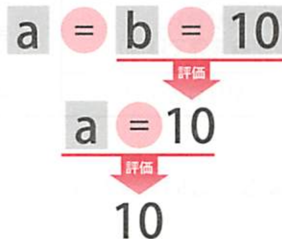




図 2-6　結合規則に従って評価される

それぞれの演算子に定められている「優先順位」と「結合規則」については次節で紹介しよう。

# 2.4 演算子

＋や＊といった演算子が出てきましたが、他にはどんな演算子があるんですか？

Javaには多くの演算子が定められているよ。その中から代表的なものを紹介していこう。

## 2.4.1 算術演算子

左右の数値オペランドを使って四則計算を行うための演算子は、**算術演算子**と総称されています。以下の5つを覚えておきましょう。

表2-3　代表的な算術演算子

| 演算子 | 機能 | 優先順位 | 評価の方向 | 評価の例 |
|---|---|---|---|---|
| + | 加算（足し算） | 中（5） | 左→右 | 3 + 5 → 8 |
| - | 減算（引き算） | 中（5） | 左→右 | 10 − 3 → 7 |
| * | 乗算（掛け算） | 高（4） | 左→右 | 3 * 2 → 6 |
| / | 除算（割り算）<br>（※整数演算では商） | 高（4） | 左→右 | 3.2 / 2 → 1.6<br>9 / 2 → 4 |
| % | 剰余（割り算の余り） | 高（4） | 左→右 | 9 % 2 → 1 |

注意が必要なのは除算演算子（/）です。この演算子は割り算を行うものですが、**整数同士の割り算に用いると「商」**を計算します。「9 / 2」が4と評価されてしまうのが困る場合は、「9.0 / 2」のようにどちらかのオペランドを小数にします。

除算演算子の落とし穴には十分、注意してほしい。

## 2.4.2 文字列結合演算子

左右の文字列オペランドを結合して１つの文字列にする演算子です。加算演算子と同じ＋記号を使います。

表 2-4　文字列結合演算子

| 演算子 | 機能 | 優先順位 | 評価の方向 | 評価の例 |
|---|---|---|---|---|
| + | 文字列の連結 | 中 (5) | 左→右 | "こん"+"にちは"→"こんにちは"<br>"ベスト"+3→"ベスト3" |

これも第１章から使ってきた演算子ですよね？

そうだね。ただし、文字列以外との結合については少し注意点があるから、2.5.4 項で詳しく説明するよ。

## 2.4.3 代入演算子

右オペランドの内容を左オペランドの変数に代入する演算子です。演算や結合を行いながら代入を行うものもあります。いずれも優先順位が最低なので「代入は基本的に最後に行われる」と覚えておけばよいでしょう。

表 2-5　代表的な代入演算子

| 演算子 | 機能 | 優先順位 | 評価の方向 | 評価の例 |
|---|---|---|---|---|
| = | 右辺を左辺に代入 | 最低 (15) | 右→左 | a = 10 → a ( 中身は 10) |
| += | 左辺と右辺を加算して左辺に代入 | 最低 (15) | 右→左 | a += 2 → a<br>(a = a + 2 と同じ ) |
| -= | 左辺から右辺を減算し左辺に代入 | 最低 (15) | 右→左 | a -= 2 → a<br>(a = a - 2 と同じ ) |
| *= | 左辺と右辺を乗算し左辺に代入 | 最低 (15) | 右→左 | a *= 2 → a<br>(a = a * 2 と同じ ) |
| /= | 左辺と右辺を除算し左辺に代入 | 最低 (15) | 右→左 | a /= 2 → a<br>(a = a / 2 と同じ ) |
| %= | 左辺と右辺を除算し、その余りを左辺に代入 | 最低 (15) | 右→左 | a %= 2 → a<br>(a = a % 2 と同じ ) |
| += | 左辺の後に右辺を連結して代入 | 最低 (15) | 右→左 | a += "風" → a<br>(a = a+"風" と同じ) |

たとえば変数aの内容を3増やしたい場合は「a += 3」と「a = a + 3」の2種類の書き方があり、どちらを用いても構いません。

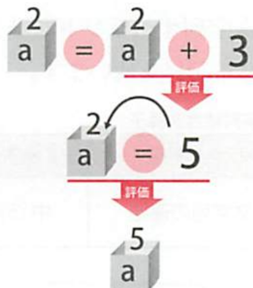


図2-7　「a = a + 3」の評価の流れ

「a = a + 3」という数式に少し違和感があるけど、評価の流れを考えれば納得できるね。

## 2.4.4　インクリメント・デクリメント演算子

前節で紹介したように、変数aの内容を3増やしたい場合には「a = a + 3」、あるいは「a += 3」と書くことができます。しかし、3ではなく1だけ増やしたり減らしたりする場合には、さらに便利な記述方法が準備されています。

表2-6　インクリメント・デクリメント演算子

| 演算子 | 機能 | 優先順位 | 評価の方向 | 評価の例 |
|---|---|---|---|---|
| ++ | 値を1増やす | 最高(1) | 左→右 | a++ → a<br>(a = a + 1 や a += 1 と同じ) |
| -- | 値を1減らす | 最高(1) | 左→右 | a-- → a<br>(a = a − 1 や a -= 1 と同じ) |

リスト2-4　インクリメントとデクリメント

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a;
4      a = 100;
```

```
5         a++;        ← aの内容が1増える
6         System.out.println(a);
7     }
8 }
```



実行結果

```
101
```

この演算子は左右両方にはオペランドを持たないんですね。

そうだね。1つしかオペランドを持たない演算子は、ほかにもあって「**単項演算子**」と総称されているよ。

## ++ や -- は、ほかの演算子と一緒に使わない！

インクリメント・デクリメント演算子は「++a」のようにオペランドの前に付けることもできます。前または後、どちらの表記法を利用しても変数aの中身が1増えることには違いありません。しかし、ほかの演算子と一緒に利用すると ++a と a++ では微妙な違いが生じます。

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int a = 10;
4     int b = 10;
5     System.out.println(++a + 50);
6     System.out.println(b++ + 50);
7   }
8 }
```

**実行結果**

```
61
60
```

変数 a も変数 b も初期値は 10 です。これに 1 を足し、50 を加えた値を表示するよう指示していますが、結果が異なっている点に注意してください。この動作の違いを理解するために、5 行目と 6 行目が次のように実行されるかを見てみましょう。

**● 5 行目の実行のされ方**

**①変数 a の値が 1 増える。**

**②それに 50 を加えたものが画面に表示される。**

**● 6 行目の実行のされ方**

**①変数 b に 50 を加えたものが画面に表示される。**

**②変数 b の値が 1 増える。**

このように他の演算と組み合わせた場合、インクリメント・デクリメント演算子がオペランドの前にあるか後にあるかで「1 増える（1 減らす）タイミング」が変わってきます。そのため、これをほかの演算子と一緒に使うと不要なバグの原因になります。特別な理由がない限り、リスト 2-4 のように単独で使うように心がけましょう。

# 2.5 型の変換



## 2.5.1 3種類の型変換

演算子の型って「いいかげん」なのかなって思うんですけど…。

確かに…。整数をdouble型変数に代入できちゃうし、「文字列と数字」を＋で結合できちゃうし…。

これは式評価の過程で「型変換」というしくみが働いてくれているおかげなんだ。

前節で学んだ演算子の多くは、原則として左右のオペランドが同じ型であることを要求します。しかし、実際には「違う型に代入」や「違う型同士で計算」をさせても文法エラーにならないことがあります。そのためJavaは型について「いいかげん」に解釈して動いてくれているように感じることもあるでしょう。

```
double d = 3;
String s = "ベスト" + 3;
```

- double d = 3; → double型変数にint型の3を代入できてしまう
- String s = "ベスト" + 3; → String型とint型を連結できてしまう

このような記述がエラーにならないのは、**Javaが式を評価する過程で自動的に型を変換している**からです。Javaには型を変換するしくみが3つ備わっていて、特に次の①と③はプログラマが気にしなくても自動的に機能します。

①代入時の自動型変換
②明示的な型変換
③演算時の自動型変換

次項から型変換のしくみについて1つずつ紹介していきます。

### 2.5.2 代入時の自動型変換

第1章の変数の解説で触れたように、**ある型で宣言された変数には、その型の値しか代入できません**（1.3.1項）。int型変数にはint型の整数だけ、String型の変数にはString型の文字列だけしか代入できない、これが原則です。

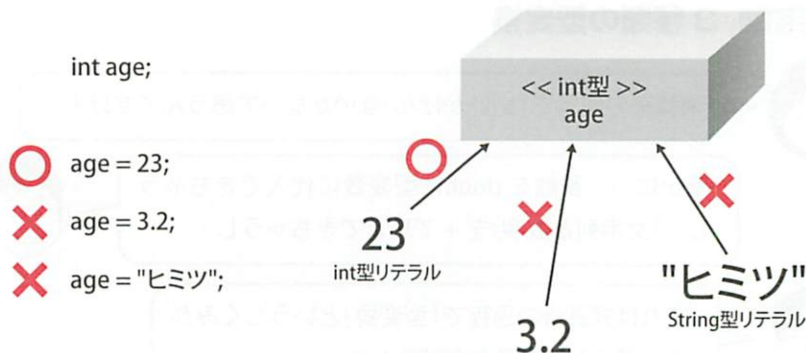



図2-8　変数の型と値の型が一致しないと代入できない

たとえばlong型の値はint型変数には代入できない。数字が大きすぎて、箱に入りきれないかもしれないしね。

でも先輩、逆にint型の値をlong型変数に入れるのは、intよりもlongの箱のほうが大きいから実害ないんじゃないでしょうか？

いいことに気づいたね。そのとおりだよ。

Javaの数値型は次の図2-9のように意味的な大小関係が定められています。そして、「小さな型」の値を「大きな型」の変数に代入する場合に限って、**値が自動的に箱の型に変換されて代入**されます。

このしくみがあるため、リスト2-5のような代入はコンパイルエラーになりません。

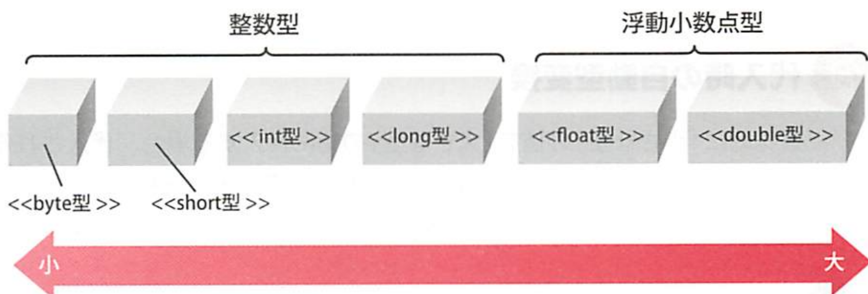



図 2-9　各数値型の意味的な大小関係



### リスト 2-5　値より大きな型の変数に代入

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     float f = 3;
4     double d = f;
5     System.out.println(f);
6     System.out.println(d);
7   }
8 }
```

float 型の変数に int 型を代入

double 型の変数に float 型を代入

**実行結果**

```
3.0
3.0
```

リスト 2-5 の 3 行目では、リテラルの 3（int 型）は 3.0F（float 型）に自動的に変換されて変数 f に代入されています。同様に 4 行目も、float 型の変数 f が double 型に変換されてから変数 d に代入されます。

## 代入時の自動型変換

意味的に「小さな型」の値を「大きな型」の箱に代入する場合、代入される値が代入先の変数の型に自動的に変換されてから代入が行われる。

逆に「大きな型」の値を「小さな型」の変数に代入することは原則としてできません。箱に入りきらない可能性があるからです。リスト2-6をコンパイルすると、「精度が落ちている可能性」という文法エラーが表示されてコンパイルは失敗します。

**リスト2-6　データより小さな型の変数に代入（エラー）**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int i = 3.2;
4    }
5  }
```

小数点以下はどうなっちゃうの？

ただし、byte型やshort型の変数に数値リテラル値を代入できないと困るため、「byte b = 3;」のように**int型リテラルをbyte型やshort型の変数に対して実害がない範囲で単純代入すること**だけは例外的に認められています。この例外を含め、ここまで学んだ代入が可能かどうかを一覧表にまとめたものが図2-10です。

図2-10　数値型に関する代入の可否

| 代入する値の型 ＼ 代入先の変数の型 | byte | short | int | long | float | double |
|---|---|---|---|---|---|---|
| byte | ◎ | ○ | ○ | ○ | ○ | ○ |
| short | × | ◎ | ○ | ○ | ○ | ○ |
| int | △ | △ | ◎ | ○ | ○ | ○ |
| long | × | × | × | ◎ | ○ | ○ |
| float | × | × | × | × | ◎ | ○ |
| double | × | × | × | × | × | ◎ |

◎……そのまま代入可能　○……自動型変換により代入可能　△……例外的・限定的に可能　×……代入できない

### 整数型としての char 型

char 型は文字を扱う型ですが、内部的には 0 ～ 65535 の範囲の数値として情報を管理しています。厳密には int や short などのような整数型の一種であるため算術演算も行えますし、型変換も行われます。しかし、一部の用途を除いて char 型を数値として利用することはほとんどないため、本書では「数値型としての char 型の取り扱い方」についての説明は割愛しました。

## 2.5.3 強制的な型変換

すでに説明したように「大きな型」の値を「小さな型」の変数に代入することは原則としてできません。しかし、それを強制的に行う方法があります。プログラマが明示的に、「小さな型に変換して押し込め！」と指示をすれば Java は変換と代入を強行します。

**リスト 2-7　強制的な型変換**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int age = (int) 3.2;
4      System.out.println(age);
5    }
6  }
```

3.2 を int に型変換して代入せよ！

**実行結果**

```
3
```

3.2 という double 型リテラルの前に記述された「(int)」が強制的な型変換を指示する**キャスト演算子**（cast operator）です。

### キャストによる強制的な型変換

( 変換先の型名 ) 式

大丈夫なんですか？ そんなことして。

もちろん大丈夫じゃない。代償を払う必要があるよ。

キャスト演算子は、**元のデータの一部を失っても**データを強制的に変換しようとします。「子ども用の小さなお弁当箱に、36cm の大判ピザを無理矢理詰め込む」ようなものですから、入りきれない部分（情報）ははみ出てしまいます。

はみ出た部分は捨てられてしまい、情報の欠損が発生します。先ほどのリスト 2-7 では、「3.2」を強引に int 型へ変換したために小数点以下の情報が失われてしまいました。

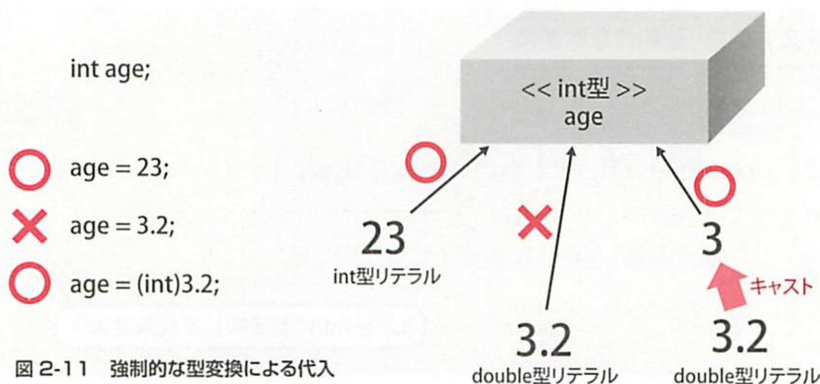



図 2-11　強制的な型変換による代入

キャストは乱暴な道具なので利用に代償を伴います。キャストを用いても変換できない型の組み合わせも存在しますし、データの欠損が不具合につながることもあります。最終手段として、どうしても必要な場合もありますが、**よほどの理由がない限り使わない**と覚えておいてください。

## 2.5.4 演算時の自動型変換

それでは、最後に3つ目の型変換のしくみを説明しよう。

代入だけではなく算術演算子などによって計算が行われる場合も「左右のオペランドは同一の型」が原則です。たとえば除算演算子 (/) による割り算のようすを見てみましょう。

図 2-12　同じ型同士で演算を行った場合

算術演算の結果は、計算で使用されたオペランドの型となります。つまり、int 型同士で計算した場合は int 型の結果、double 型同士で計算した場合は double 型の結果になります。

では、異なる型で演算を行った場合はどうなるでしょうか？ その場合には**「意味的に大きな型」に統一されてから演算**が行われます。

図 2-13　異なる型同士で演算をした場合

### 数値型同士の演算時型変換のルール

- 片方のオペランドが double なら、他方を double に型変換して揃える。
- 片方のオペランドが float なら、他方を float に型変換して揃える。
- 片方のオペランドが long なら、他方を long に型変換して揃える。
- 片方のオペランドが int なら、他方を int に型変換して揃える。
- short や byte のオペランドは int に型変換される。

代入にしても演算にしても、Javaはきっちりと型を揃えてから処理するんですね。

そうだよ。実際に演算時に型変換を行うサンプルも見ておこうか。

**リスト2-8 異なる型同士の算術演算**

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     double d = 8.5 / 2;
4     long l = 5 + 2L;
5     System.out.println(d);
6     System.out.println(l);
7   }
8 }
```

2（int型）を2.0（double型）に変換

5（int型）を5L（long型）に変換

**実行結果**

```
4.25
7
```

「intとlong」のように数値型同士の演算でなく、「intとString」のように「数値型と文字型」の組み合わせでも、別のルールで自動的に型が変換される。これも紹介しておこう。

オペランドが数値型とString型の場合は、次のような型変換が自動的に行われます。

## 文字列を含む演算時型変換

片方のオペランドがStringなら、他方もStringに変換して連結する。

**リスト 2-9　文字列の連結**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String msg = "私の年齢は" + 23;
         23 (int 型) が "23" (String 型) に変換されて連結される
4      System.out.println(msg);
5    }
6  }
```

**実行結果**

```
私の年齢は23
```

おめでとう、これで「②計算の文」はすべてマスターしたよ。最後の「③命令実行の文」に進もう。

## Java言語仕様をのぞいてみよう

本書では初めて Java を学ぶ人にもわかりやすいように文を 3 種類に分類して解説しています。しかし、Java における文の厳密な分類はとても複雑です。Java の正式な決まりは「Java 言語仕様（The Java Language Specification）」にまとめられています。書籍や Web サイトで閲覧できますので、ぜひ参照してください。

# 2.6 命令実行の文

## 2.6.1 命令実行の文とは

「②計算の文」は演算子とか型変換とか、たくさん出てきて結構、複雑でした…。

お疲れさま。でも喜んでほしい。最後の1つ「③命令実行の文」はとてもカンタンで、しかも楽しいよ。

ここで図1-14（p.43）を、もう一度見てください。ここまでJavaにおける3種類の文のうち2種類を解説してきました。最後に残っているのは「③命令実行の文」です。

命令実行の文は**Javaが準備してくれているさまざまな命令を呼び出す**ための文です。この文を使えば、「足し算」や「代入」より、ずっと高度な処理をコンピュータに行わせることができます。最も代表的なものとして、おなじみの「System.out.println」があります。

**リスト2-10　命令（画面出力）実行の文**

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     String name = "すがわら";                ①変数宣言の文
4     String message;
5     message = name + "さん、こんにちは";      ②計算の文
6     System.out.println(message);            ③命令実行の文
7   }
8 }
```

**実行結果**

```
すがわらさん、こんにちは
```

ちなみに、リスト 2-10 は 5 行目と 6 行目を 1 つの文にまとめて、「System.out.println(name + " さん、こんにちは ");」にもできる。

命令実行の文の中で式を使うこともできるんですね。

命令実行の文は、末尾に丸カッコで囲まれた部分が登場するのが特徴です。

**命令実行の文**

**呼び出す命令の名前 ( 引数 );**

カッコの中に記述するものは**引数**や**パラメータ**と呼ばれるもので、その命令を呼び出すにあたって必要となる追加情報です。System.out.println() であれば、「何を画面に表示するか」という情報を引数で指定します。

Java で利用できる命令は System.out.println() 以外にも数多くありますが、引数を 2 つ指定するものや、1 つも指定しなくてもよいものなど、それぞれ引数の種類や数が異なります。

ほかにはどんな命令があるんですか？
一発でゲームが作れちゃうような命令とかあるんですか？！

まぁまぁ落ち着いて。少しずつ紹介していくから、
楽しみにしていてほしい。

使える命令が System.out.println() だけでは楽しくありませんね。Java には「音

を鳴らす」「ファイルに書き込む」「キーボードから入力を受け付ける」「プリンタに文字を印刷する」「ネットワークで通信を行う」など、数多くの命令が準備されています。しかし、現時点の私たちには、それらすべてを使いこなすことは難しいので、次項以降では使いやすい命令を少しずつ紹介していきます。

なお、紹介した命令について書き方を丸暗記する必要はありません。後から使いたくなったときに「そういえばこういう命令があったはず」と思い出して本書を読み返せば大丈夫です。気楽に読み進めてください。

## 2.6.2 画面に文字を表示する命令

まずは基礎中の基礎、画面に文字を表示する命令からいこう。System.out.println() と似た命令をもう１つ紹介しよう。

**改行せずに画面に文字を表示する**

```
System.out.print( ① );
```

※①は画面に表示したい値や式

System.out.println() 命令とよく似ている命令に System.out.print() 命令があります。この命令は画面に①の内容を表示しますが、表示後に改行しません。このため、連続して呼び出すと表示内容が連続して表示されます。

**リスト 2-11　改行なし画面出力の命令**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String name = "すがわら";
4      System.out.print("私の名前は");
5      System.out.print(name);
```

```
6       System.out.print("です");
7     }
8   }
```

実行結果

私の名前はすがわらです

## 2.6.3 大きいほうの数字を代入する命令

### 2つの値を比較して大きいほうの数字を代入する

```
int m = Math.max ( ①, ② );
```

※①および②は比較したい値や式

Math.max() 命令は、2つの引数を指定して呼び出す命令です。引数として与えた①と②のうち、大きなほうの値が m に代入されます。なお、解説の都合で変数名として m を使っていますが、ほかの変数名でも構いません。

リスト 2-12　大きいほうの数字を選択させる命令

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a = 5;
4      int b = 3;
5      int m = Math.max(a, b);
6      System.out.println("比較実験：" +
           a + "と" + b + "とで大きいほうは…" + m );
7    }
8  }
```

**実行結果**

```
比較実験：5と3とで大きいほうは…5
```

「Math.max ( 5,3 ) 」の結果が 5 になっているんですね。

「Math.max ( 5,3 )」が 5 に化ける…。これ、「評価」ですか？

鋭いね。実は「命令の実行」も式の一種なんだ。

## 2.6.4 文字列を数字に変換する命令

**文字列を数字に変換する**

```
int n = Integer.parseInt ( ① );
```

※①は数字として解釈させたい文字列（"23" など）

たとえばString型変数に入っている "10" は文字列なので、そのままでは四則演算ができません。**文字列の "10" を数字の 10 に変換して計算を行いたい**場合には、この命令を使ってください。

命令の①に「整数として読むことができる」文字列が入ったString型の変数やリテラルを指定すると、int型の整数に変換してnに代入してくれます。

**リスト 2-13　String型をint型に変換する命令**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String age = "31";
```

```
4      int n = Integer.parseInt(age);
5      System.out.println
           ("あなたは来年、" + ( n + 1 ) + "歳になりますね。");
6    }
7  }
```

**実行結果**

あなたは来年、32歳になりますね。

もし①に「こんにちは」のような「数字ではない文字列」を指定すると、プログラム実行中にエラーが起きて異常終了するから気をつけてほしい。

## 2.6.5 乱数を生み出して代入する命令

湊くんが大好きなゲームに不可欠なのが乱数だ。

ランスウ……？

コンピュータの中に入っているサイコロみたいなものよ。毎回ランダムに違う値が取り出せるの。

**乱数を発生させる**

```
int r = new java.util.Random().nextInt( ① );
```

※①は発生させる乱数の上限値（指定値自体を含まない）

①に1以上の整数を指定してこの命令を呼び出すと、0以上かつ①で指定した数字未満のランダムな整数がrに代入されます。rに何が代入されるかは実行するまでわかりません。①に10を指定するとrには0～9のいずれかが代入されます。

**リスト2-14　ランダムな数を生成する命令**

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int r = new java.util.Random().nextInt(90);
4      System.out.println("あなたはたぶん、" + r + "歳ですね？");
5    }
6  }
```

**実行結果**

```
あなたはたぶん、31歳ですね？
```

## 2.6.6 キーボードから1行の入力を受け取る命令

あとゲーム作りに必要なのは「キーボードから文字を入力する」命令だね。

**キーボードから1行の文字列の入力を受け付ける**

```
String input = new java.util.Scanner ( System.in ).
nextLine ();
```

**キーボードから1つの整数の入力を受け付ける**

```
int input = new java.util.Scanner ( System.in ).
nextInt ();
```

これらの文を実行すると、プログラムは一時停止状態になり、利用者がキーボードから文字を入力できるようになります。そして、利用者が文字列をキーボード入力してEnterキーを押すと、その内容が変数inputに代入されます。nextLine()は文字列を、nextInt()は数字の入力を受け取るために使います。



**リスト2-15　キーボードから入力を受け付ける命令**

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("あなたの名前を入力してください。");
4     String name = new java.util.Scanner(System.in).nextLine();
5     System.out.println("あなたの年齢を入力してください。");
6     int age = new java.util.Scanner(System.in).nextInt();
7     System.out.println
          ("ようこそ、" + age + "歳の" + name + "さん");
8   }
9 }
```

**実行結果**

```
あなたの名前を入力してください。
すがわら⏎        ← キーボードから名前を入力
あなたの年齢を入力してください。
31⏎              ← キーボードから年齢を入力
ようこそ、31歳のすがわらさん
```

# 2.7 第2章のまとめ

この章では、次のようなことを学びました。

## 式

- 式は演算子とオペランドで構成されている。
- リテラルにも型があり記述方法で決定される。
- 演算子が評価されると、その演算子とオペランドは結果に化ける。
- 演算子は優先順位と結合規則に従い評価される。

## 型変換

- 大きい変数に小さなデータを代入する際、自動的に型が変換され代入される。
- 小さな変数に大きなデータを代入する際、キャストを行うことで代入できる。
- 式の評価時、大きなデータに揃えるよう自動的に型が変換される。

## 命令の実行

- Javaに用意されている、さまざまな命令を実行することができる。

# 2.8 練習問題

## 練習 2-1

次のようなプログラムがあります。

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int x = 5;
4     int y = 10;
5     String ans = "x+yは" + x + y;
6     System.out.println(ans);
7   }
8 }
```

このプログラムを実行すると以下の結果が表示されます。

```
x+yは510
```

「x+y は 15」と表示させたいのですが、意図どおりに動きません。正しく動作するように修正してください。

## 練習 2-2

次の中で文法として正しいものを、すべて選んでください。

① int x = 3 + 5.0;
② double d = 2.0F;
③ int i = "5";
④ String s = 2 + " 人目 ";
⑤ byte b = 1;
⑥ double d = true;
⑦ short s = (byte)2;

### 練習 2-3

以下の内容のプログラムを作成してください。

①画面に「ようこそ占いの館へ」と表示します。
②画面に「あなたの名前を入力してください」と表示します。
③キーボードから1行の文字入力を受け付け、String型の変数nameに格納します。
④画面に「あなたの年齢を入力してください」と表示します。
⑤キーボードから1行の文字入力を受け付け、String型の変数ageStringに格納します。
⑥変数ageStringの内容をint型に変換し、int型の変数ageに代入します。
⑦0から3までの乱数を生成し、int型の変数fortuneに代入します。
⑧fortuneの数値をインクリメント演算子で1増やし、1から4の乱数にします。
⑨画面に「占いの結果が出ました！」と表示します。
⑩画面に「(年齢)歳の(名前)さん、あなたの運気番号は(乱数)です」と表示します。その際に(年齢)には変数ageを、(名前)には変数nameを、そして(乱数)には⑧で作った数字を表示させます。
⑪画面に「1: 大吉 2: 中吉 3: 吉 4: 凶」と表示します。

# 2.9 練習問題の解答

### 練習 2-1 の解答

5 行目を次のように修正します。

```
String ans = "x+yは" + (x + y);
```

不具合の原因は 5 行目の最後の「x + y」を丸カッコ「()」で囲っていなかったからです。「2.5.4 演算時の自動型変換」で解説したように、オペランドの中に文字列が含まれると、そのほかのオペランドも文字列型に変換されます。そのため int 型の変数である x と y の内容は文字列型に変換され「文字列として連結」されます。その結果、画面に「x+y は 510」と表示されたのです。

意図どおりに「x+y は 15」と表示させるには、x と y を丸カッコで囲い、この計算の評価順位を引き上げる必要があります。

### 練習 2-2 の解答

正しい文は②、④、⑤、⑦です。

## 練習 2-3 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("ようこそ占いの館へ");
4      System.out.println("あなたの名前を入力してください");
5      String name = new java.util.Scanner(System.in).nextLine();
6      System.out.println("あなたの年齢を入力してください");
7      String ageString =
           new java.util.Scanner(System.in).nextLine();
8      int age = Integer.parseInt(ageString);
9      int fortune = new java.util.Random().nextInt(4);
10     fortune++;
11     System.out.println("占いの結果が出ました！");
12     System.out.println(age + "歳の" + name +
           "さん、あなたの運気番号は" + fortune + "です");
13     System.out.println("1: 大吉  2:中吉  3:吉  4: 凶");
14   }
15 }
```