6章 データの型変換を理解しよう

Javaにおけるデータの型変換について学び、実践します。

60分 - 未読

6.1 本章の目標

本章では以下を目標にして学習します。

- Javaにおけるデータの型変換について知り、実践できること

プログラミングでは、さまざまな種類のデータを扱います。データ型が異なるデータ同士を扱う場合に、知っておくべきテクニックが「型変換」です。

本章では、Javaにおける型変換の基本から使い方まで、実践しながら学びます。データの達人を目指して、Javaの型変換をしっかり覚えましょう。

なお本章の実践パートでは、**パッケージ「text.section_06」を作成**してください。その中に**ファイル「TypeConv_番号.java」** を順番に作成しましょう（作成方法は3章を参照）。

「TypeConv」は、type conversion（型変換）を略したものです。

6.2 型変換とは

本節では、型変換の基本を学びます。ただし、そもそもデータ型を理解していないと、型変換も理解できません。まずはデータ型について復習しましょう。

データ型について復習しよう

データ型とは、整数の「123」や文字列の「こんにちは」といったデータの種類でしたね。3章では9種類のデータ型を学びました。一覧表で、今一度チェックしましょう。

分類	データ型	値の範囲	サイズ
整数型	byte	-128～127	8ビット
整数型	short	-32,768～32,767	16ビット
整数型	int	-2,147,483,648～2,147,483,647	32ビット

+ 質問する

分類	データ型	値の範囲	サイズ
整数型	long	-9,223,372,036,854,775,808～9,223,372,036,854,775,807	64ビット
浮動小数点型	float	(-3.4028235 × 10の38乗)～(3.4028235 × 10の38乗)	32ビット
浮動小数点型	double	(-1.7976931348623157 × 10の308乗)～(1.7976931348623157 × 10の308乗)	64ビット
論理型	boolean	true または false	1バイト※ 環境による
文字型	char	0～65,535　※文字コードを表す	2バイト
文字列型	String	任意の文字列（文字の集まり）	※環境による

上記のように、データ型によって値の性質も、範囲も大きく異なります。そのためデータ型を取り違えると、正しくデータを扱えない場合があるのです。

たとえば「1.0」というdouble型の小数を、整数用のint型に代入（値をセット）しようとすると、以下のようにエラーとなります。データ型が一致せず、上手く扱えないためです。

型変換とは

上記のようなデータ型の不一致による問題を解決してくれるのが、型変換です。型変換とは、あるデータ型の値を、別のデータ型へ変換すること。

たとえば、以下のようなシチュエーションで型変換が必要となります。

- データ型の異なる変数へ値をコピーするとき（例：long型の値をint型にコピー）
- 計算によりデータ型が変わるとき（例：整数を整数で割った結果が小数になる）
- ファイルから取り込んだ文字列を、数値として計算に使いたいとき

Javaの型変換は、大きく分けて**自動型・手動型**の2種類です。

種類	説明
自動型変換	コンパイラ（プログラムを機械向けに翻訳するソフトウェア）が自動でデータ型を変換する
手動型変換	プログラマーが後述の「キャスト演算子」により明示的にデータ型を変換する

要は、プログラマーが明確に書くべき型変換と、書かなくてよい型変換があります。すべての型変換が自動で行われるわけではないため、ときには手動による型変換が必要です。

型変換を安易に行うと、思わぬ形にデータが変わり、プログラムの誤動作を招く恐れがあります。いつ型変換が必要なのかを理解し、**適切なタイミングで行うことが大切**です。

次節からは、自動型変換・手動型変換（キャスト演算子）の順にそれぞれ学びましょう。

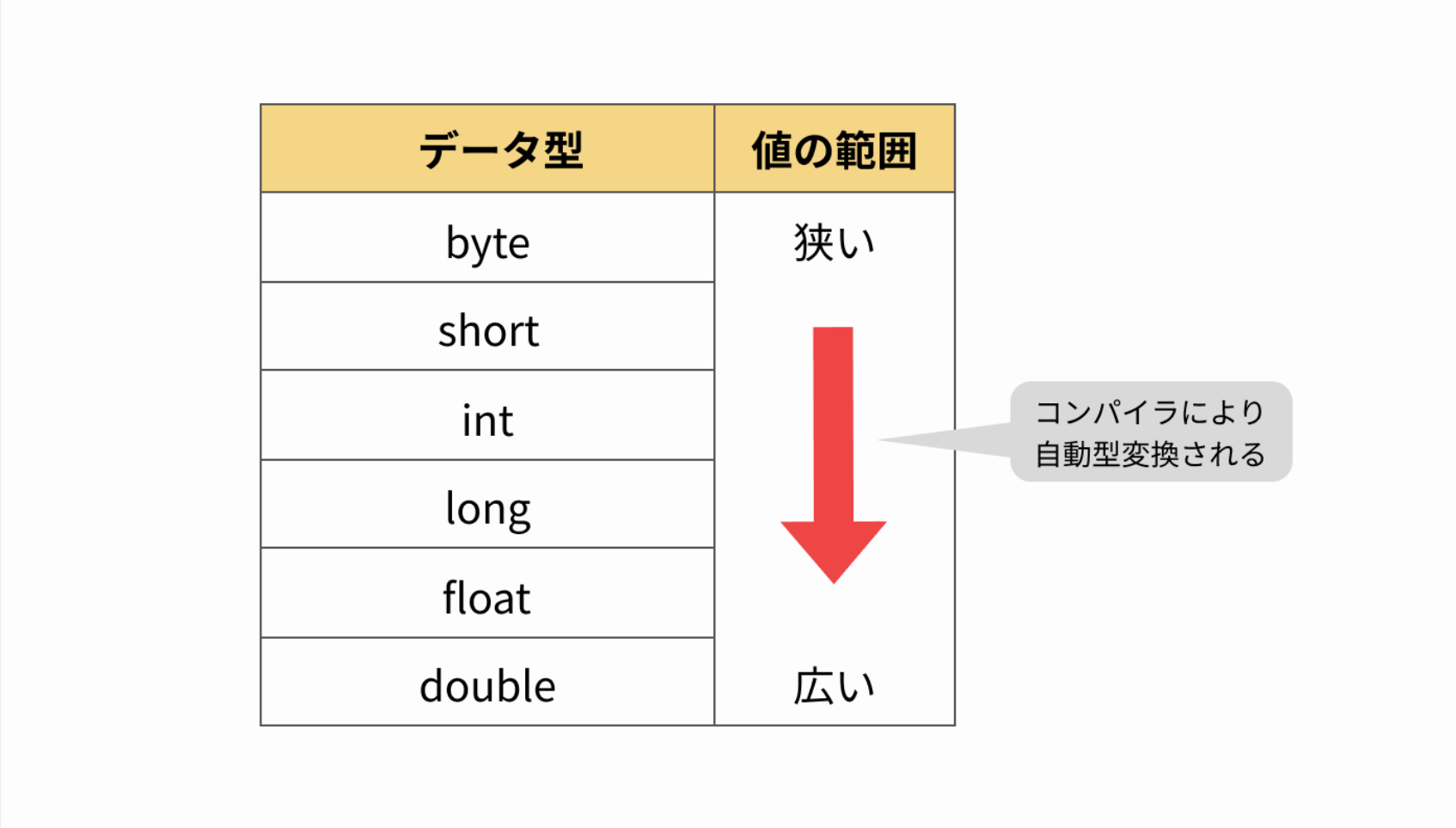
6.3 自動型変換

自動型変換は、プログラマーが明示しなくても自動的にデータ型を変換してくれるものです。

Javaは、実行前にプログラム全体の変換が必要な「コンパイラ型言語」だと1章で学びました。自動型変換は、**コンパイラがソースコードを変換するとき**に行われます。

自動型変換が行われる条件

コンパイラによる自動型変換が行われるのは、**「より広範囲なデータ型への変換」のみ**に限られます。表のデータ型でいえば、上の型から下の型へ変換するケースです。



小さい箱に収まる物は、大きい箱にもそのまま収まりますよね。同じように、範囲の狭いデータ型から広範囲なデータ型に変える場合は、データが壊れる心配がありません。

たとえば、以下ではint型の「123」を、より広範囲なlong型の変数「valLong」に代入しています。123のような整数のリテラル（値）がint型扱いとなるのは、4章で学びましたね。

```
long valLong = 123; // int型の値をlong型へ型変換してから代入

```

123の代入先（valLong）は、100億を超える値でも簡単に扱えるlong型です。int型の123を入れたところでデータが壊れる心配が全くないため、自動型変換の対象となります。

このように、自動型変換は**データ型が変わっても状態を保てる場合**に行われます。反対に、データが壊れる恐れがある場合は、プログラマー自身の判断による型変換が必要です。

自動型変換の流れ

前項のソースコードをコンパイラが翻訳するとき、以下のように型変換を行います。

1. 代入先のlong型にあわせて、「123」をlong型へ型変換する
2. long型となった「123」を、変数 `valLong` に代入する

適切な型変換をコンパイラが行ってくれるため、エラーが出ることなく実行できるのです。では、ファイル「TypeConv_1.java」を作成して、実際に試してみましょう。

TypeConv_1.java

```
1  package text.section_06;
2
3  public class TypeConv_1 {
4      public static void main(String[] args) {
5
6          // [範囲の狭いデータ型]→[広範囲なデータ型]の自動型変換
7          long   valLong   = 123;     // int型の値をlong型に変換して代入
8          float  valFloat  = 123;     // int型の値をfloat型に変換して代入
9          double valDouble = 123.4F; // float型の値をdouble型に変換して代入
10
11         System.out.println(valLong);
12         System.out.println(valFloat);
13         System.out.println(valDouble);
14     }
15 }
16
```

上記のプログラムでは、以下の自動型変換をともなう変数の初期化を行っています。

- int型→long型
- int型→float型
- float型→double型

Eclipseも自動型変換に対応しているため、特にエラーなく実行できるはずです。実行して、以下のように表示されることを確認してください。

```
1  123
2  123.0
3  123.4000015258789
4
```

2行目が「123.0」となっていますが、これは浮動小数点型のデータですね。「123」がfloat型に変換されているため、小数点を含む形で表示されているのです。

3行目も同様に、double型に変換されたデータが表示されています。「123.4」から若干ずれているのは、浮動小数点型の誤差によるものです。

浮動小数点型は誤差が生じる場合があると、4章で学びました。**float型やdouble型に変換する場合も、誤差は発生し得る**ため注意しましょう。

6.4 キャスト演算子

自動で行われない型変換は、プログラマーが行わなければなりません。明示的に型変換を行うときに使うのが、**キャスト演算子**です。基本的な書き方は、以下のとおり。

```
(変換したいデータ型)[型変換したい変数や値]

```

キャスト演算子は**データ型を括弧ではさみ**、**型変換したい変数や値の頭につける**のが基本です。たとえば、変数「test」をキャスト演算子でint型へ変換する場合、以下のように書きます。

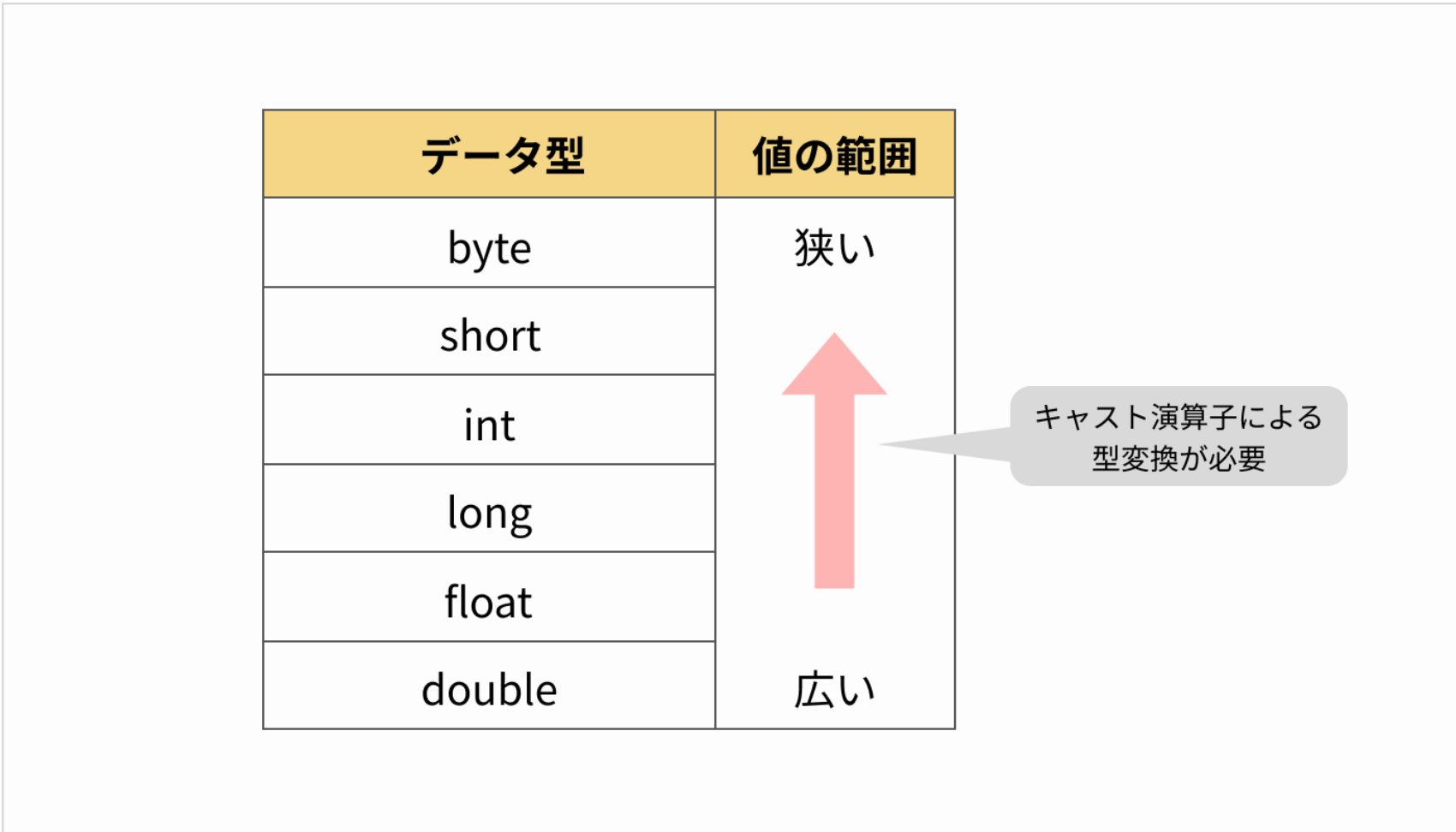
```
(int)test

```

なお、「キャスト」は型変換の別名です。プログラマーがキャスト演算子を使って型変換することを、「キャストする」などと表現することもあります。

キャスト演算子による型変換が必要となる条件

キャスト演算子による明示的な型変換は、**「より範囲が狭いデータ型への変換」** で必要となります。表のデータ型でいえば、下の型から上の型へ変換するケースです。



大きい箱に入っていた物が、小さい箱に収まるとは限りません。広範囲なデータ型から範囲の狭いデータ型に無理やり置き換えれば、データが壊れる恐れがあります。

たとえば、short型変数の値をbyte型変数に代入する場合、以下のようにエラーとなります。byte型は-128～127の整数しか扱えず、short型の値が入らない恐れがあるためです。

```
short valShort = 123;
byte  valByte  = valShort;
```

型の不一致: short から byte には変換できません

仮にshort型の最大値「32,767」をbyte型変数に無理やり代入すれば、不正な値に変わってしまうでしょう。それどころか、近くにある別のデータを壊してしまう恐れもあります。

このように、より範囲の狭いデータ型への変換はリスクが高く、コンパイラが自動で行うわけにはいかないのです。そのため、**キャスト演算子による型変換が必要かどうかは、プログラマーが判断**しなければなりません。

キャスト演算子の注意点

キャスト演算子で型変換を行う場合、**思わぬ形にデータが変わり誤動作を招くリスク**があります。範囲の狭いデータ型への変換が必要だからと、安易に使うべきではありません。

例として、以下のようにソースコードを書いてみましょう。

TypeConv_2.java

```
package text.section_06;

public class TypeConv_2 {
    public static void main(String[] args) {

        // short型の値をbyte型にキャスト
        short valShort = 32767;
        byte  valByte  = (byte)valShort; // byte型に収まらず不正値となる
        System.out.println(valByte);

        // double型の値をint型にキャスト
        double valDouble = 123.456;
        int    valInt    = (int)valDouble; // 小数部分はカットされる
        System.out.println(valInt);
    }
}

```

キャスト演算子による以下2種類の型変換を行い、結果を表示するプログラムです。

- short型の「32767」をbyte型にキャスト
- double型の「123.456」をint型にキャスト

では、プログラムを実行してみましょう。エラーは出なくても、以下のように元々のデータと結果が変わってしまいますね。

```
-1
123

```

範囲を超えた値をキャストする場合、1行目のように不正値となります。また、浮動小数点型を整数型にキャストする場合、2行目のように小数部分はカットされます。

このように、キャスト演算子で範囲の狭いデータ型へ変換する場合、値の一部が失われたり不正値となったりします。型変換を行う場合、こうしたリスクを念頭に置きましょう。

キャスト演算子による型変換が役に立つ場面

「いつキャスト演算子を使えばいいの？」と疑問に思いますよね。ここでは、キャスト演算子による型変換が役に立つ場面を紹介します。

小数点以下まで正確に計算したいとき

キャスト演算子によって、小数をともなう計算を正確に行える場合があります。たとえば、距離と所要時間から速度を計算する以下のプログラムは、速度が正しく表示されません。

```
int distance = 120; // 距離(m)
int time = 25; // 所要時間(秒)

// 距離と所要時間から速度(m/秒)を計算
float speed = distance / time;
System.out.println("速度：" + speed + "m/秒");

```

距離120mを所要時間25秒で割れば、結果は4.8m/秒のはずです。しかし、距離と所要時間を表す変数（distance、time）がともにint型のため、計算結果もint型として扱われます。

計算結果は小数を含む「4.8」でも、整数しか扱えないint型に自動型変換が行われてしまうのです。それにより小数部分がカットされて、以下のように不正確な結果が表示されます。

```
速度：4.0m/秒

```

この場合は、キャスト演算子を使うことで正しく計算できるようになります。実際に、以下のようにソースコードを書いて、試してみましょう。

TypeConv_3.java

```
package text.section_06;

public class TypeConv_3 {
    public static void main(String[] args) {

        int distance = 120; // 距離(m)
        int time = 25; // 所要時間(秒)
        // 距離と所要時間から速度(m/秒)を計算
        float speed =  (float)distance / (float)time;
        System.out.println("速度：" + speed + "m/秒");
    }
}

```

以下の箇所では、変数 `distance` と変数 `time` を浮動小数点型のfloat型にキャストしています。こうすることで計算結果もfloat型として扱われ、小数部分がカットされないのです。

```
float speed = (float)distance / (float)time;

```

上記のプログラムを実行すると、以下のように正確な速度が表示されます。

```
速度：4.8m/秒

```

このように、計算結果の小数部分を正確に扱いたい場合、キャスト演算子が役に立ちます。とはいえ上記例では、距離と所要時間の変数がそもそもfloat型なら、キャストは不要です。

しかし、ソフトウェアの仕様や設計上、変数のデータ型を自己判断で変えられないケースは考えられます。その場合は、こうしたキャスト演算子による型変換が役に立つでしょう。

小数点以下を切り捨てたいとき

先ほどとは反対で、必要なのは整数にもかかわらず、計算結果に小数が生じてしまう場合もあります。たとえば、商品の税抜価格から税込価格を計算するようなケースです。

こうしたケースでも、キャスト演算子が役に立ちます。実際に試してみましょう。以下のようにソースコードを書いてください。

TypeConv_4.java

```
package text.section_06;

public class TypeConv_4 {
    public static void main(String[] args) {

        int     price = 298; // 価格(円)
        double    tax = 0.1; // 消費税(%)

        // 税込価格(円)を計算
        int charge = (int) (price + (price * tax) );
        System.out.println("料金：" + charge + "円");
    }
}

```

これは、298円の商品における税込価格を計算するプログラムです。消費税10%で計算すると298+29.8=327.8円となりますが、価格表示で小数部分は不要ですよね。

そこで、以下のように計算結果をint型にキャストしています。int型が整数しか扱えないことを逆手にとって、不要な小数部分を切り捨てるのです。

```
int charge = (int) (price + (price * tax) );

```

上記のキャスト演算子は、「price + (price * tax)」という計算式全体に適用されます。計算結果をキャストしたい場合は、このように括弧で計算式をまとめましょう。

このプログラムを実行すると、不要な小数部分が切り捨てられて、正しい税込価格が表示されます。

```
料金：327円

```

このように小数部分を切り捨てたい場合は、整数型にキャストするのが1つの手段です。

本章の学習は以上です。お疲れさまでした。

まとめ

本章では以下の内容を学習しました。

- 用語
  - 型変換（キャスト）：あるデータ型の値を、別のデータ型へ変換すること
- 文法
  - 型変換は自動型・手動型の2種類
    - 自動型変換：より広範囲なデータ型への変換
      - コンパイラがソースコードを変換するときに自動で行う
    - 手動型変換：より範囲の狭いデータ型への変換
      - プログラマーがキャスト演算子を用いて明示的に行う
        - (変換したいデータ型)[型変換したい変数や値] のように記述
      - 思わぬ形にデータが変わり誤動作を招くリスクがある
      - キャスト演算子が役に立つ場面
        - 小数点以下まで正確に計算したいとき
        - 小数点以下をカットしたいとき

型変換は、データや変数を使いこなすために知っておくべき知識です。適切なタイミングで型変換を行えば、正確な計算ができるなどのメリットがあります。

一方で型変換は、プログラムの誤動作の原因になりやすい要素でもあります。正確なプログラムを作るために、型変換についてしっかり理解を深めておきましょう。

次章では、条件分岐のif文について学びます。

理解度を選択して次に進みましょう

ボタンを押していただくと次の章に進むことができます

～50%　50～80%　80～100%

最後に確認テストを行いましょう

下のボタンを押すとテストが始まります。

教材をみなおす　テストをはじめる

© SAMURAI Inc.　利用規約　法人会員利用規約　プライバシーポリシー　運営会社