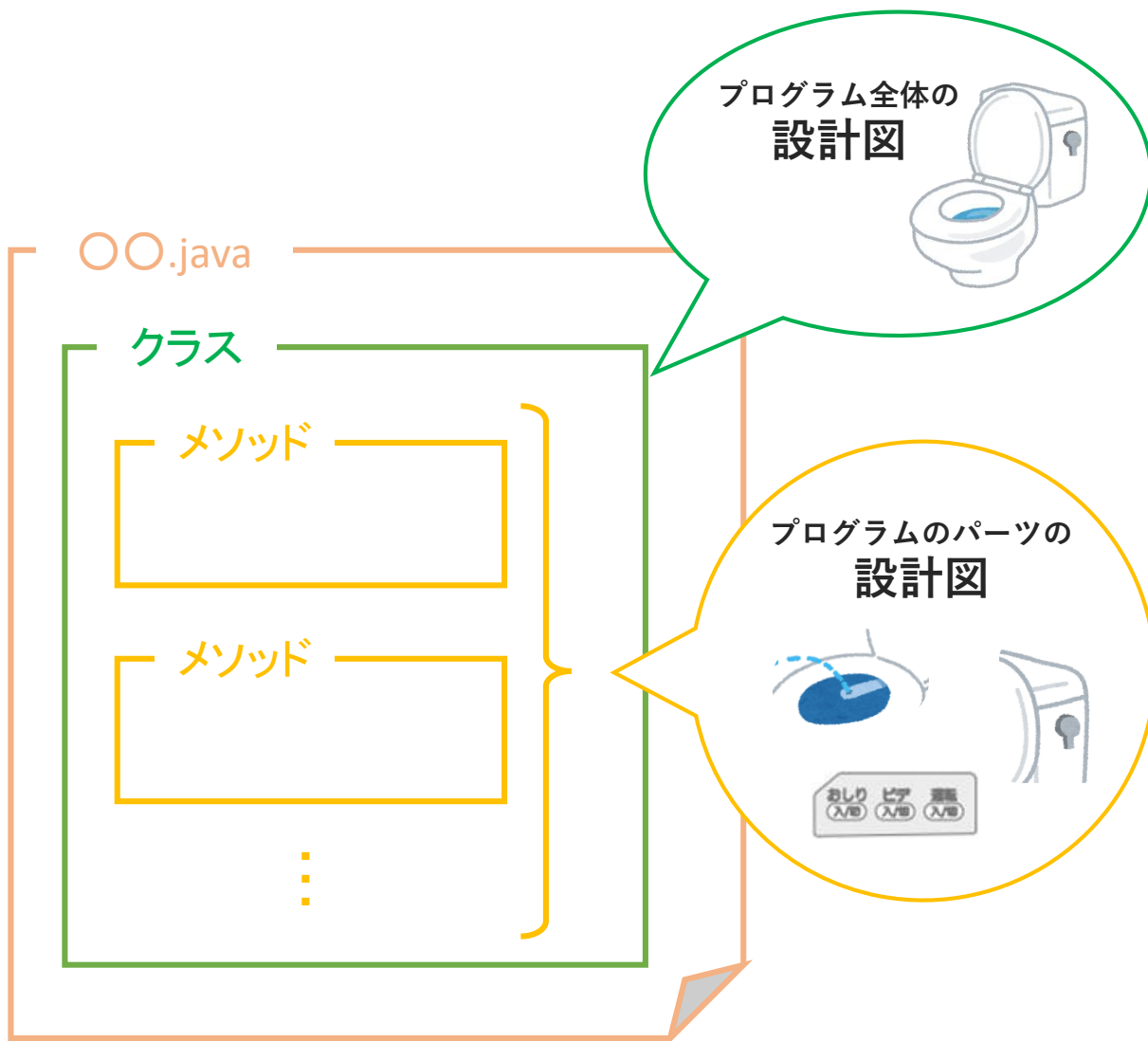


The background features a grayscale profile of a man's head facing left. Overlaid on his hair is a circular arrangement of code snippets in various programming languages, including Python, JavaScript, and Java. The main title is written in large, bold, black Japanese characters across the center of the image.

ウズウズカレツジ プログラマーコース

クラスとメソッド

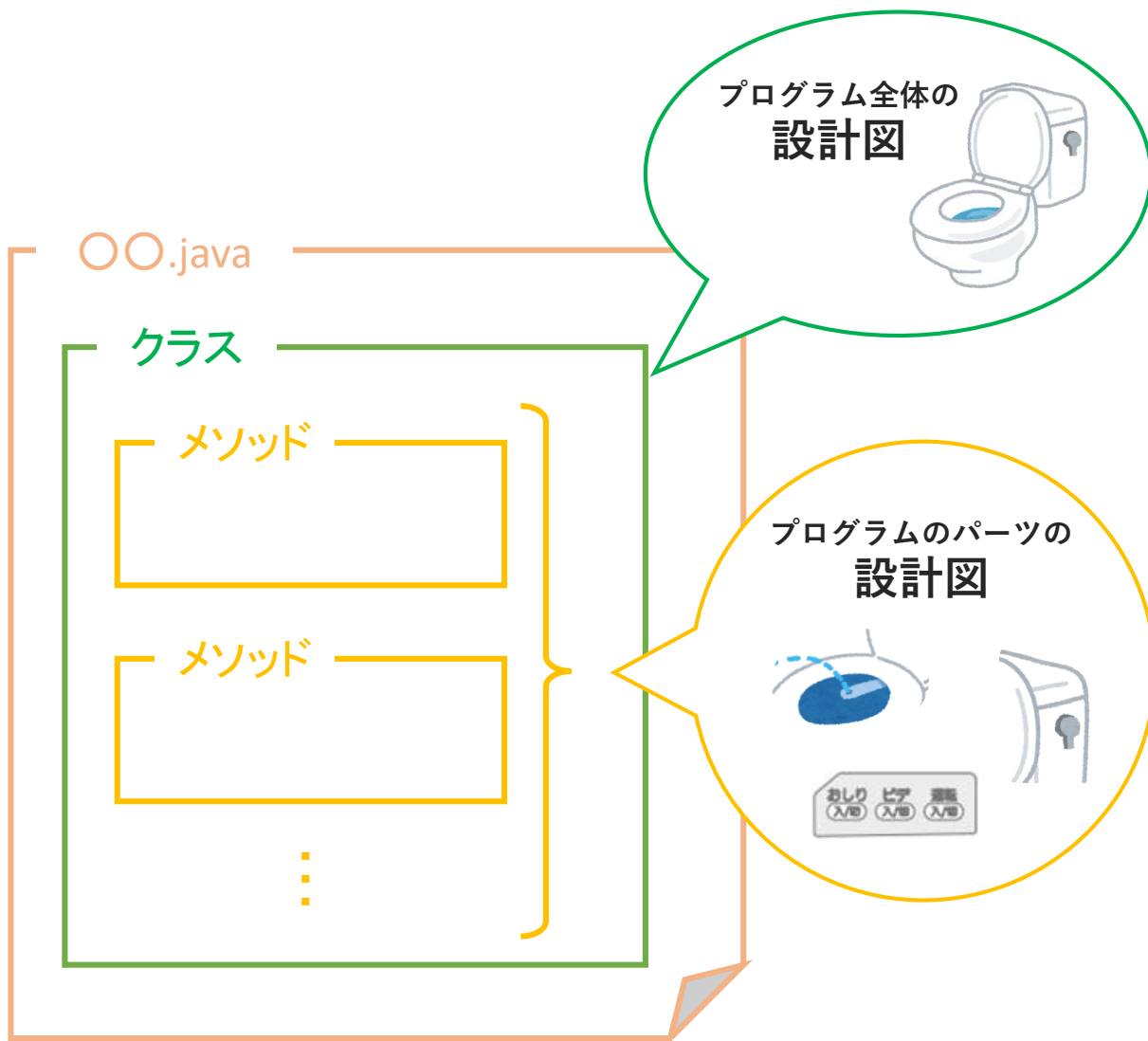
はじまる!!



《クラス》

- クラスはプログラム全体の**設計図**です。
クラスブロックで囲われた領域がこのプログラムで実施したい具体的な処理内容であることを表します。
- 基本的に **1つのJavaファイルには1つのクラス**だけ定義します。
- **ファイル名とクラス名は同じ**であるように定義します。
- クラスブロックは下記のように定義します。

```
class クラス名 { }
```
- クラス名の命名規則は下記のとおりです。
 - ・ **先頭を大文字**
 - ・ **それ以外は小文字**
 - ・ **言葉の区切りは大文字**



《メソッド》

□メソッドはプログラムの**パーツ（一部機能）の設計図**です。
メソッドブロックで囲われた領域でパーツがどのように動くかを記述します。
メソッドのことを**関数**と呼ぶこともあります。

□基本的に**1つの機能ごとに1つのメソッド**を作成します。

□1つのクラスに複数メソッドを定義することは可能で、
クラスブロックの直下に並べて書いていきます。
(メソッドブロックの中にメソッドブロックは定義できません)

□メソッドブロックは下記のように定義します。

[修飾子] 戻り値の型 メソッド名(引数の型 引数名, ...) { 命令群 }

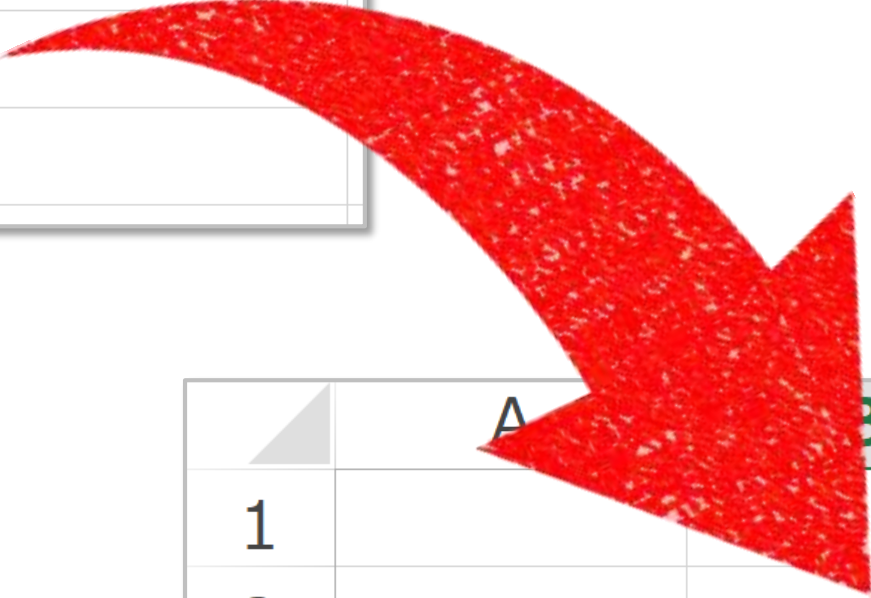
※サンプルソースコードで修飾子として記述されている「static」や「public」については、今は「無ければエラーになる」くらいの感覚で大丈夫です。

□メソッド名の命名規則は下記のとおりです。
変数の命名規則と基本的に同じです。

- ・先頭を小文字
- ・以降も小文字
- ・言葉の区切りは大文字
- ・予約語（後述）ではない英単語と略語を組み合わせることが多い

～Excelの関数のイメージを持とう！～

	A	B	C
1			
2		=sum(2,3)	
3			



	A	B	C
1			
2			5
3			

～Excelの関数のイメージを持とう！～

《Excelの表》

《Excelの裏》

引数

sum(2, 3)

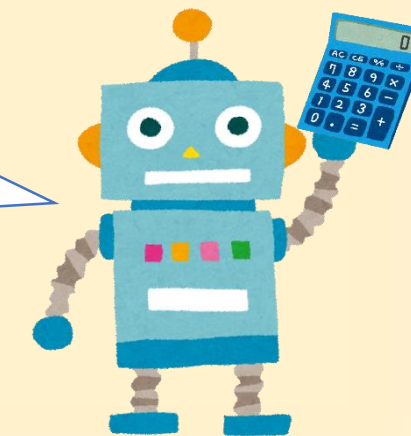
sum君
このデータ
よろしく！

仮引数

2

3

はい！
受け取りました！



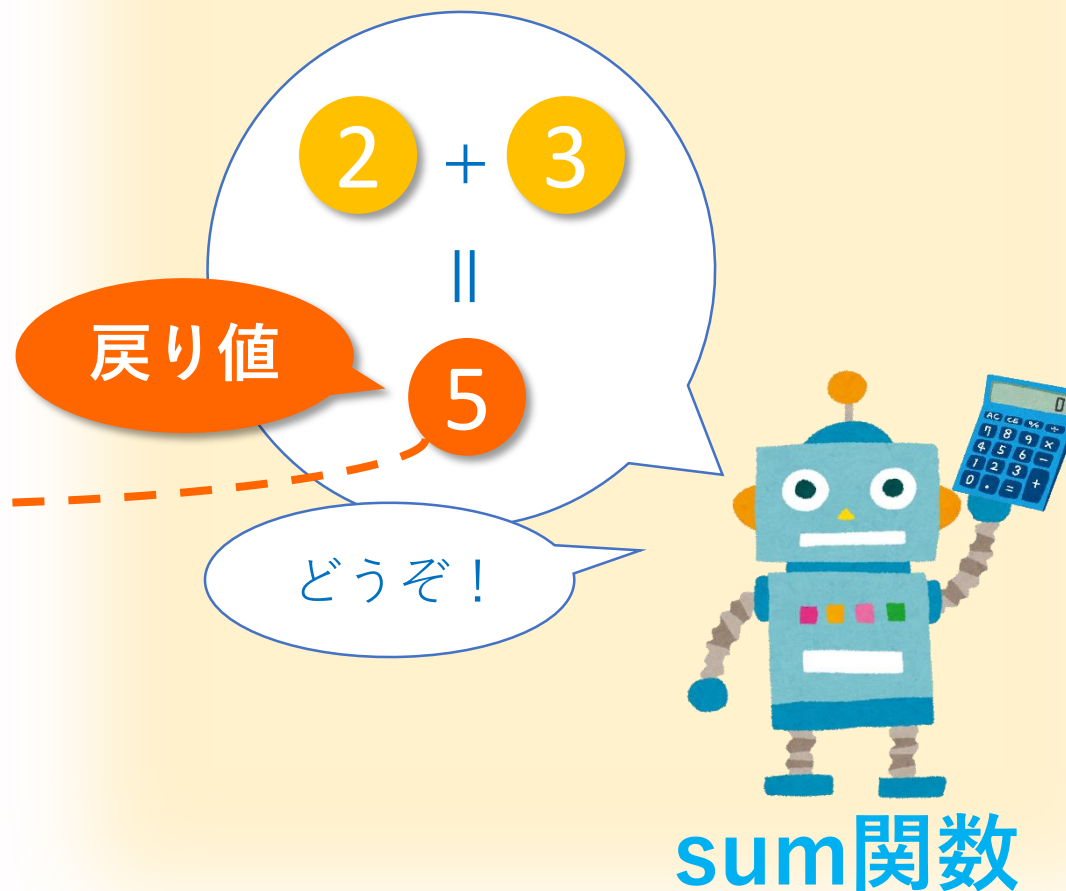
sum関数

～Excelの関数のイメージを持とう！～

《Excelの表》



《Excelの裏》

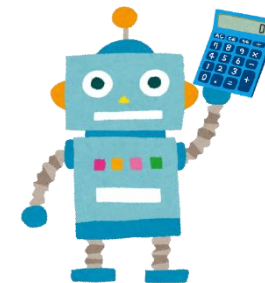


～Excelの関数のイメージを持とう！～

```
1 class Sample1_14_1 {  
2 ^  
3 ^ public static void main(String[] args) {  
4 ^ ^  
5 ^ ^ //メソッドのしくみ  
6 ^ ^  
7 ^ ^ int print = sum( 2 , 3 ) ;  
8 ^ ^  
9 ^ ^ System.out.println("print : " + print);  
10 ^ ^  
11 ^ ^ }  
12 ^  
13 ^ static int sum( int num1 , int num2 ) {  
14 ^ ^  
15 ^ ^ int calcResult = num1 + num2 ;  
16 ^ ^  
17 ^ ^ return calcResult ;  
18 ^ ^  
19 ^ ^ }  
20 ^  
21 }
```

Sample1_14_1クラス

mainメソッド



sumメソッド

～メソッドの構造を理解しよう！～

戻り値の型の左側

メソッド名の左隣

() の左隣

() の中

修飾子

戻り値の型

メソッド名

仮引数の宣言

```
static int sum( int num1 , int num2 ) {  
    int calcResult = num1 + num2 ;  
    return calcResult ;  
}
```

命令群

return文

戻り値を返す

戻り値

戻り値を返す

return文 の後

～メソッドの動きを理解しよう！～

START

```
public static void main(String[] args) {  
    //メソッドのしくみ  
    int print = sum( 2 , 3 ) ;  
    System.out.println("print : " + print);  
}
```

print

～メソッドの動きを理解しよう！～

START

```
public static void main(String[] args) {  
    ^  
    ^ //メソッドのしくみ  
    ^  
    ^ int print = sum( 2 , 3 ) ;  
    ^  
    ^ System.out.println("print : " + print);  
    ^  
    ^  
    ^ }
```

sumメソッドの
呼び出し

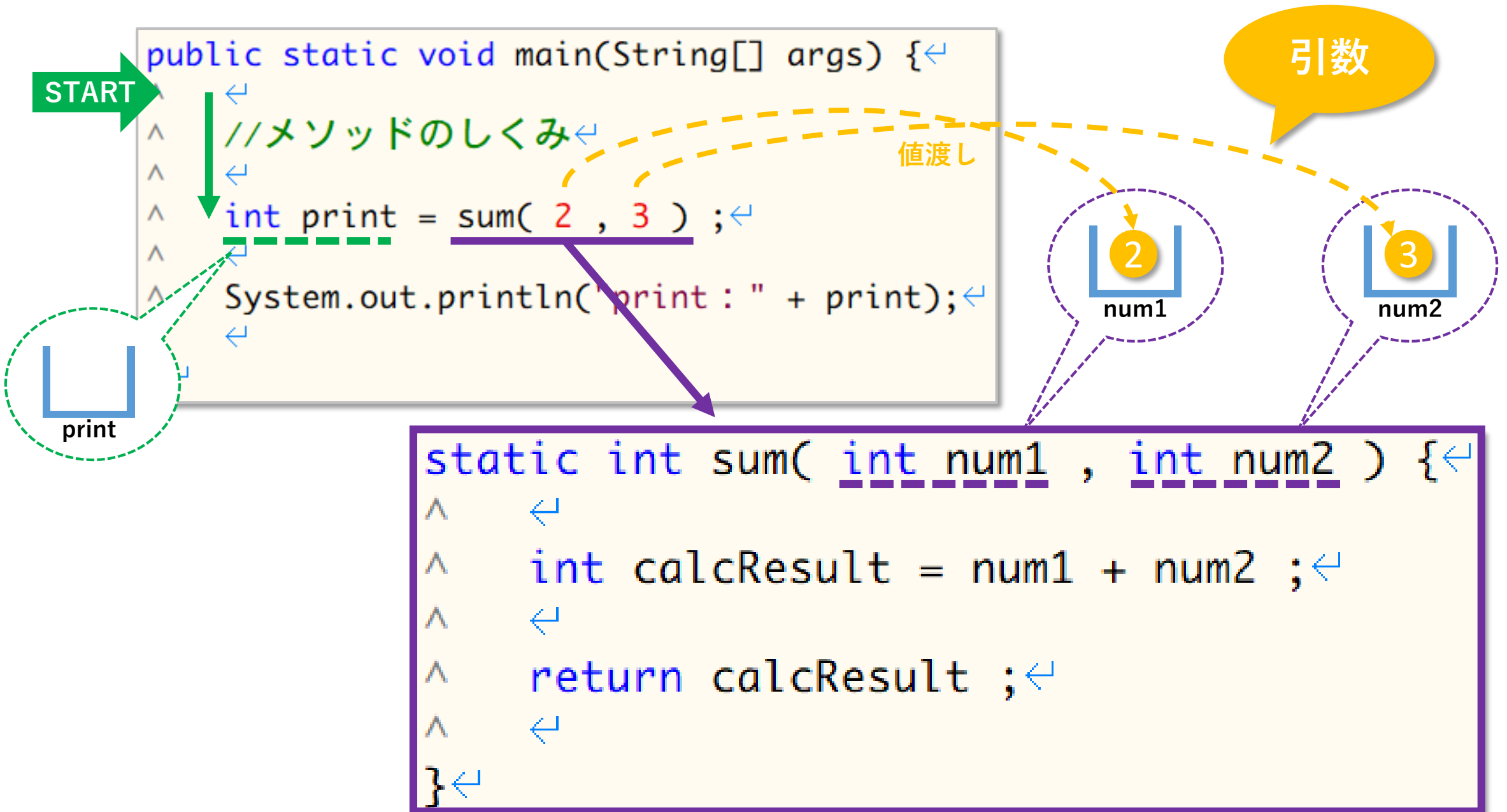
print

num1

num2

```
static int sum( int num1 , int num2 ) {  
    ^  
    ^ int calcResult = num1 + num2 ;  
    ^  
    ^ return calcResult ;  
    ^  
    ^  
    ^ }
```

～メソッドの動きを理解しよう！～



～メソッドの動きを理解しよう！～

START

```
public static void main(String[] args) {  
    ^  
    ^ //メソッドのしくみ  
    ^  
    ^ int print = sum( 2 , 3 ) ;  
    ^  
    ^ System.out.println("print : " + print);  
    ^  
    ^  
}
```

print

2

num1

3

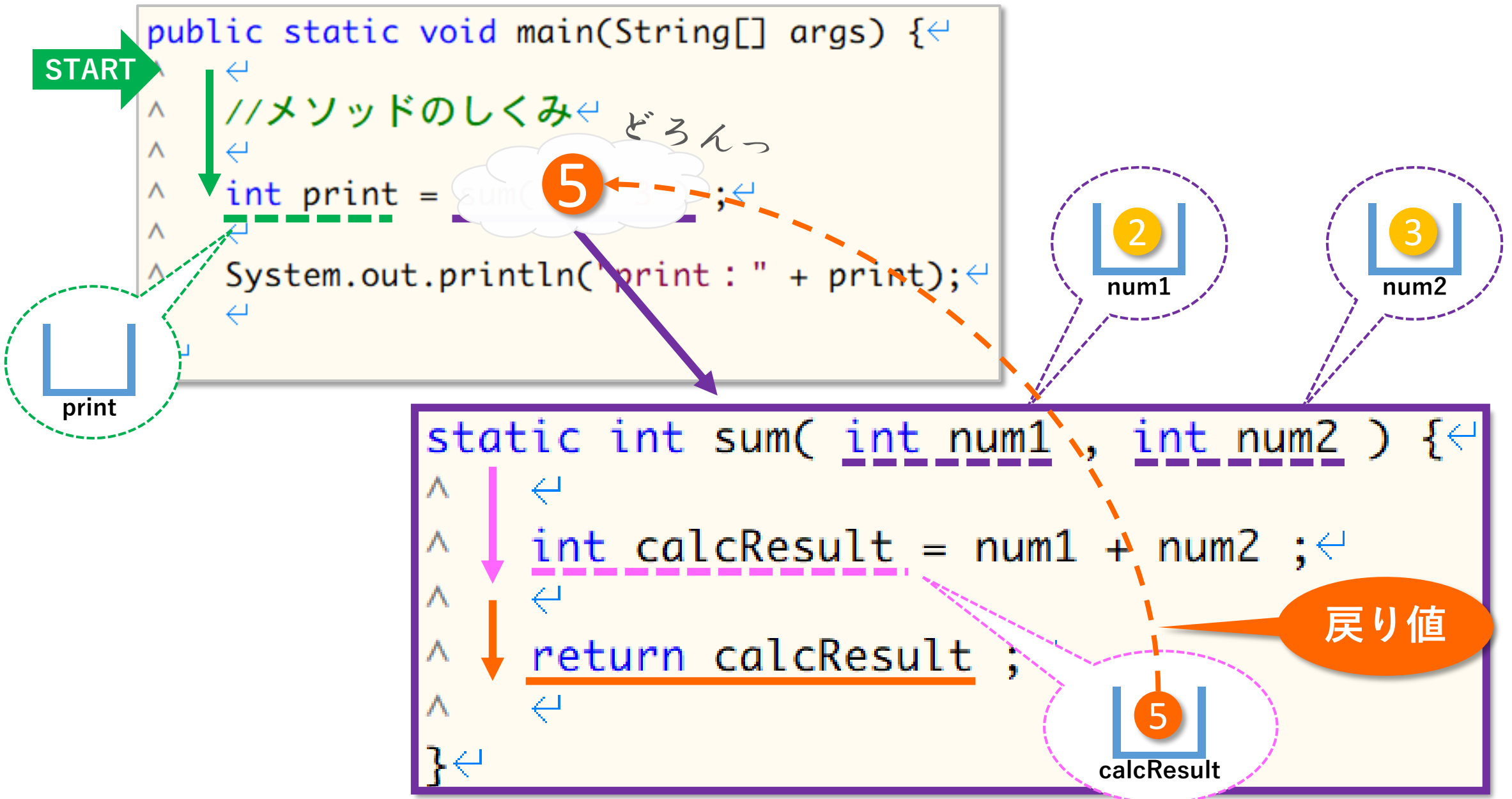
num2

```
static int sum( int num1 , int num2 ) {  
    ^  
    ^ int calcResult = num1 + num2 ;  
    ^  
    ^ return calcResult ;  
    ^  
    ^  
}
```

5

calcResult

～メソッドの動きを理解しよう！～



～メソッドの動きを理解しよう！～

START

```
public static void main(String[] args) {  
    //メソッドのしくみ  
    int print = sum(3, 2);  
    System.out.println("print : " + print);  
}
```

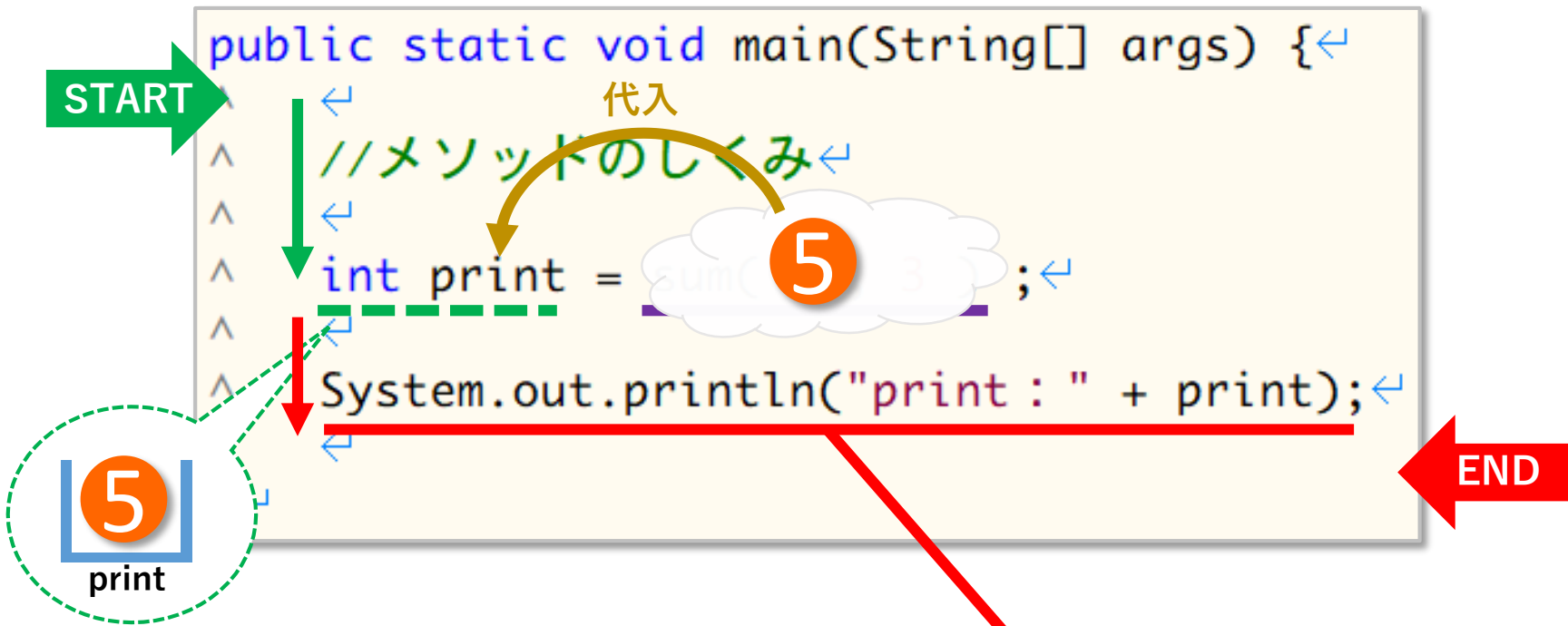
代入

5

5

print

～メソッドの動きを理解しよう！～



```
C:\¥Workspace>java Sample1_14_1  
print : 5
```

```

1 class Sample1_14_2 {
2     ^
3     //引数が不要なメソッド
4     static String getName() {
5         ^
6         return "モコ" ;
7         ^
8     }
9     ^
10    //戻り値なしのメソッド
11    static void printName(String name) {
12        ^
13        System.out.println("なまえ：" + name);
14        ^
15    }
16    ^
17    //プログラムは必ずmainメソッドからはじまる
18    public static void main(String[] args) {
19        ^
20        String print = getName() ;
21        ^
22        printName( print );
23        ^
24    }
25    ^
26 }

```

引数が不要な
メソッド

戻り値のない
メソッド

《引数のないメソッド / 戻り値のないメソッド》

- メソッドの中には**引数が不要なメソッド**や**戻り値のないメソッド**が存在します。
- 引数が不要なメソッドは下記のように仮引数を書かずに定義すればOKです。

[修飾子] **戻り値の型** **メソッド名()** { **命令群** }

- 戻り値のないメソッドは「戻り値の型」を書くべき箇所に「**void**」と記述すれば「戻り値が存在しない」という意味になります。

[修飾子] **void** **メソッド名(引数の型 引数名, ...)** { **命令群** }

sum(2 , 3)

引数 : int型 2つ

sum(1.2 , 1.3)

引数 : double型 2つ

sum(2 , 3 , 4)

引数 : int型 3つ

▼Sample1_14_3.java (21~46行目)

```
//sum (int型の引数2つ) ←
static int sum( int num1 , int num2 ) {←
^   ←
^   int calcResult = num1 + num2 ;←
^   ←
^   return calcResult ;←
^   ←
}←
←
//sumメソッド (double型の引数2つ) ←
static double sum( double num1 , double num2 ) {←
^   ←
^   double calcResult = num1 + num2 ;←
^   ←
^   return calcResult ;←
^   ←
}←
←
//sum (int型の引数3つ) ←
static int sum( int num1 , int num2 , int num3 ) {←
^   ←
^   int calcResult = num1 + num2 + num3 ;←
^   ←
^   return calcResult ;←
^   ←
}←
```

《オーバーロード》

- メソッドはメソッド名だけでなく「引数の型」や「引数の数」でも区別されます。
逆にメソッド名が合致していても引数の型や引数の数が違っていればそのメソッドを呼び出すことはできません。
- 「引数の型」「引数の数」が違っていればクラス内で同名のメソッドを複数定義することが可能であり、この機能をオーバーロードと言います。

～printlnメソッドとオーバーロード～

▼Sample1_14_printlnクラス（printlnの呼出し元）

```
class Sample1_14_println {  
^   ^  
^   ^ public static void main(String[] args) {  
^   ^   ^  
^   ^   ^ //printlnはオーバーロードされて定義されている  
^   ^   ^  
^   ^   ^ System.out.println("▼print1 (int型) ");  
^   ^   ^ int print1 = 1 ;  
^   ^   ^ System.out.println( print1 );  
^   ^   ^  
^   ^   ^ System.out.println("▼print2 (double型) ");  
^   ^   ^ double print2 = 2.0 ;  
^   ^   ^ System.out.println( print2 );  
^   ^   ^  
^   ^   ^ System.out.println("▼print3 (char型) ");  
^   ^   ^ char print3 = '3' ;  
^   ^   ^ System.out.println( print3 );  
^   ^   ^  
^   ^   ^ System.out.println("▼print4 (String型) ");  
^   ^   ^ String print4 = "4" ;  
^   ^   ^ System.out.println( print4 );  
^   ^   ^  
^   ^   ^ System.out.println("▼print5 (boolean型) ");  
^   ^   ^ boolean print5 = true ;  
^   ^   ^ System.out.println( print5 );  
^   ^   ^  
^   ^ }  
^ }  
}
```

引数の型に応じて
最適なメソッドが
自動で選ばれる



▼printlnが定義されているクラス（PrintStreamクラス）

void println(int i){ 引数を表示する命令群 }

void println(double d){ 引数を表示する命令群 }

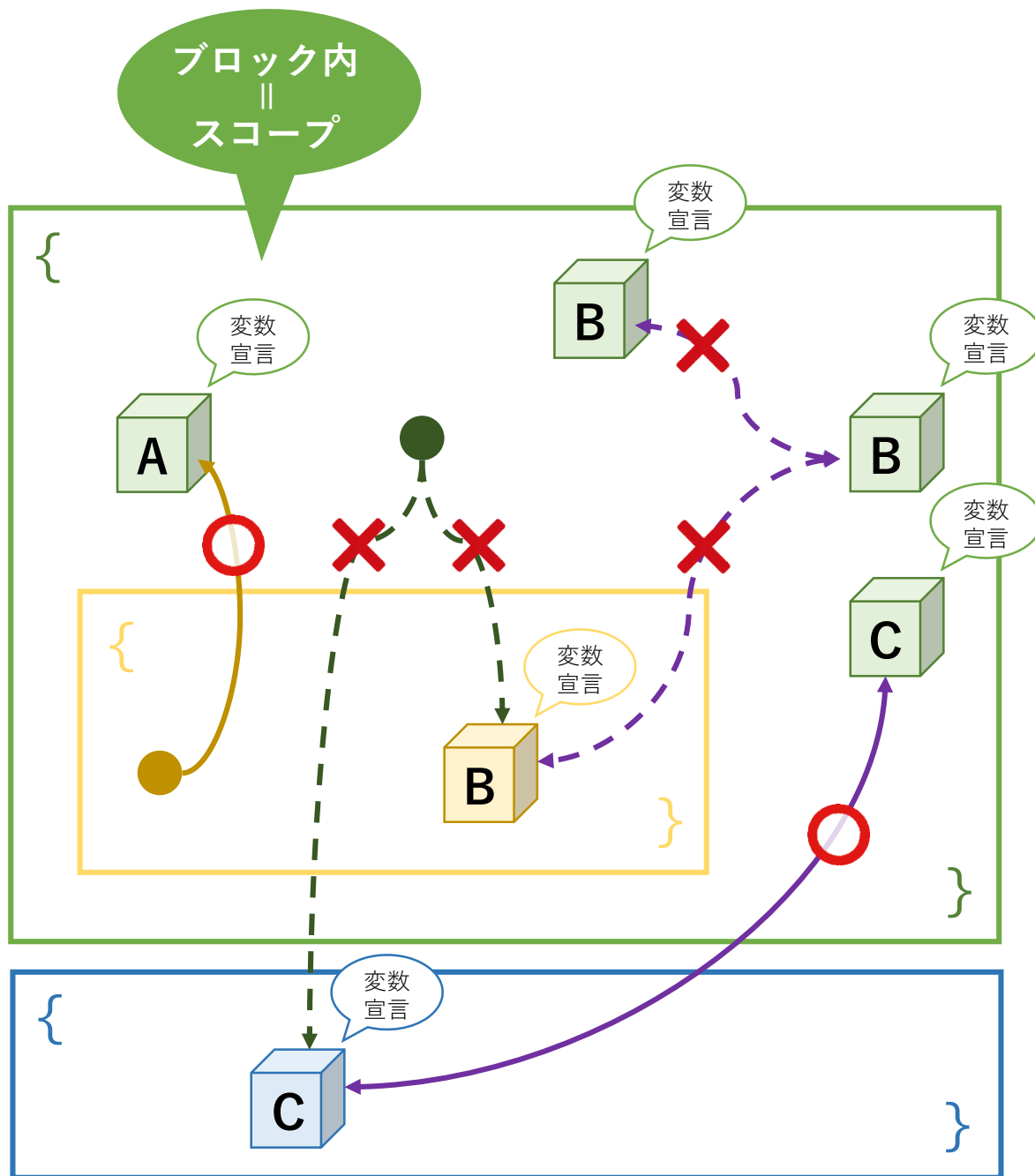
void println(char c){ 引数を表示する命令群 }

void println(String s){ 引数を表示する命令群 }

void println(boolean b){ 引数を表示する命令群 }

⋮

オーバーロードされて
定義されている

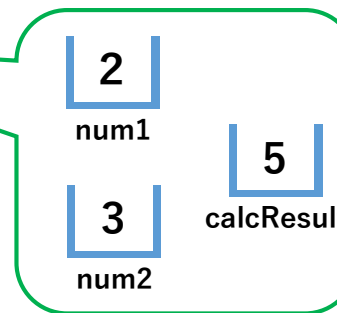
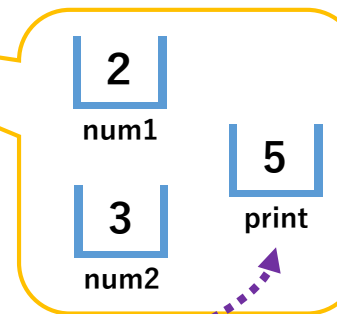
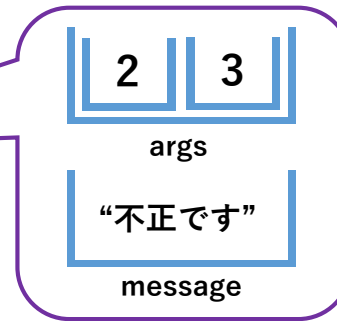


《スコープとローカル変数》

- 変数には有効範囲があり、これを**スコープ**と言います。
変数宣言されたブロック内（{ } で囲まれた範囲）がその変数のスコープとなります。
スコープ外からのアクセスはすべてエラーとなります。
- メソッド内で定義される変数を**ローカル変数**と言います。
当然スコープ外である他のメソッドからは参照したり代入したりできません。
- ブロック内の処理がすべて実行されるとそのブロックをスコープとする変数は役目を果たしたとしてメモリ上から消去されます。
- ブロックがネスト（入れ子）構造になっている場合、内側のブロックから外側で宣言された変数にアクセスすることは可能です。
ネストは関係なく、**単純に変数宣言されたブロック内すべてがその変数のスコープだと理解しましょう。**
- 特定ができないため、**同じスコープ内で同じ名前の変数を定義することはできません。**
スコープが別であれば変数名について特に気にすべきことはありません。
- { } の外での変数宣言になりますが、**メソッドの仮引数はメソッドブロック内をスコープとしていることに注意してください。**

～変数のスコープを理解しよう～

```
class Sample1_14_4 {  
    public static void main( String[] args ) {  
        //ローカル変数のしくみ  
        String message = "不正です" ;  
        if( args.length == 2 ){  
            int num1 = Integer.parseInt( args[0] );  
            int num2 = Integer.parseInt( args[1] );  
            int print = sum( num1 , num2 );  
        }else{  
            System.out.println( message );  
        }  
        System.out.println( "print : " + print );  
    }  
    static int sum( int num1 , int num2 ) {  
        int calcResult = num1 + num2 ;  
        return calcResult ;  
    }  
}
```



<演習：Ex1_14_1>

これまで使ってきた以下のメソッドがどのように定義されているか考えてみましょう

(1) Integer.parseIntメソッド

- ①戻り値の型はなんですか？
- ②仮引数の型はなんですか？（複数思い当れば思い浮かぶだけ出しましょう）
- ③オーバーロードされて定義されているでしょうか？

(2) String.valueOfメソッド

- ①戻り値の型はなんですか？
- ②仮引数の型はいくつでしょう？（複数思い当れば思い浮かぶだけ出しましょう）
- ③オーバーロードされて定義されているでしょうか？

<演習：Ex1_14_2>

```
public class Ex1_14_2 {  
    ^  
    ^ // !!! mainメソッドの処理は書き換えないでください !!! ^  
    ^ public static void main (String[] args) {  
    ^     ^  
    ^     ^ int totalPrice = 0 ; //購入した商品の合計金額^  
    ^     ^  
    ^     ^ //コマンドライン引数から購入した商品の合計金額（定価）を取得^  
    ^     ^ for(int i = 0 ; i < args.length ; i++) {  
    ^     ^     ^ totalPrice += Integer.parseInt( args[i] );  
    ^     ^     ^ }  
    ^     ^  
    ^     ^ //discountメソッドを使って割引を適用し、割引後の金額を取得^  
    ^     ^ int discountedPrice = discount( totalPrice ); //割引後の金額^  
    ^     ^  
    ^     ^ //calcTaxPaymentメソッドを使って支払金額（税込）を取得^  
    ^     ^ int taxPayment = calcTaxPayment( discountedPrice ); //支払金額（税込）^  
    ^     ^  
    ^     ^ //支払金額（税込）を表示^  
    ^     ^ System.out.println("割引後の支払金額：" + taxPayment + "円" );  
    ^     ^  
    ^ }  
    ^  
    ^ /*  
    ^ **以下の仕様を持つメソッドcalcTaxPaymentを作成してください。^  
    ^ ** - 引数として受け取った値の税込価格（消費税は8%とする）を計算して返す^  
    ^ ** - 税込価格は整数（小数点以下切り捨て）で返す^  
    ^ */  
    ^ static _____ calcTaxPrice( _____ ){ //アンダーバーを適切な内容に書き換えてください^  
    ^     ^  
    ^     ^ }  
    ^ }  
    ^  
    ^ /*  
    ^ **以下の仕様を持つメソッドdiscountを作成してください。^  
    ^ ** - 引数として受け取った値が1000以上5000以下の場合、1000より大きい分について10%割引にする^  
    ^ ** - 引数として受け取った値が5000より大きい場合、1000より大きく5000円以下の分について10%OFF、5000円より大きい分について20%割引にする^  
    ^ ** - 値引き額は小数点以下切り捨てで計算する（キャストを使いましょう）^  
    ^ ** - 割引金額の上限は5000円とする^  
    ^ */  
    ^ static _____ discount( _____ ){ //アンダーバーを適切な内容に書き換えてください^  
    ^     ^  
    ^     ^ }  
    ^ }  
}^
```

C:\¥Workspace>java Ex1_14_2 900
割引後の支払金額：972円

C:\¥Workspace>java Ex1_14_2 1000
割引後の支払金額：1080円

C:\¥Workspace>java Ex1_14_2 4600
割引後の支払金額：4579円

C:\¥Workspace>java Ex1_14_2 5000
割引後の支払金額：4968円

C:\¥Workspace>java Ex1_14_2 25000
割引後の支払金額：22248円

C:\¥Workspace>java Ex1_14_2 30000
割引後の支払金額：27000円