

# 1.3 制御フロー

- 条件分岐
- 繰り返し処理
- メソッド

# 目次

- 1 条件分岐
- 2 繰り返し処理
- 3 メソッド

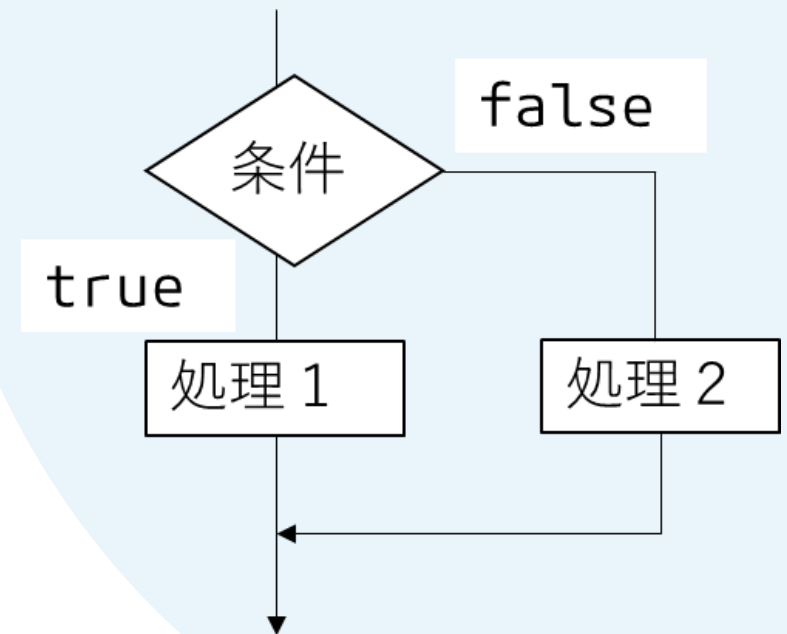
# 条件分岐

- いままで私たちが書いたプログラムは、順番に実行され、コードがスキップされたり繰り返されたりしません。
- ある条件を満たしたときだけコードを実行させたい場合は、**分岐文**[\[Branch Statement\]](#)が必要です。
- Java では、分岐の実装方法として、以下のような方法があります：
  - if-else 文；
  - switch-case 文；
  - 条件演算子で分岐文と同等の機能を実現できる。

# if 文

- 条件分岐を行うには最も基本的なのは、**if** 文です：

```
if (condition) {
    codes;
}
```



- condition はブール値である必要があります。条件が成立すると、「{ }」内のコードが実行されます。そうでない場合は、この中括弧はスキップされます。
- 練習：次のコードは何を出力しますか？

```
1 int x = 20;
2 int y = 18;
3 if (x > y) {
4     System.out.println("x is bigger: " + x);
5 }
6 System.out.println("y is bigger: " + y);
```



# else 文

- **else** 文は、if が満たされない場合の処理を記述します：

```
if (condition) {  
    codes 1;  
} else {  
    codes 2;  
}
```

- condition の値が true の場合、最初の中括弧の中のコードのみが実行されます。 false の場合、2 番目の中括弧の中のコードのみが実行されます。



- 練習：次のコードは何を出力しますか？

```
1 int time = 10;
2 if (time < 18) {
3     System.out.println("Good day.");
4 } else {
5     System.out.println("Good evening.");
6 }
```

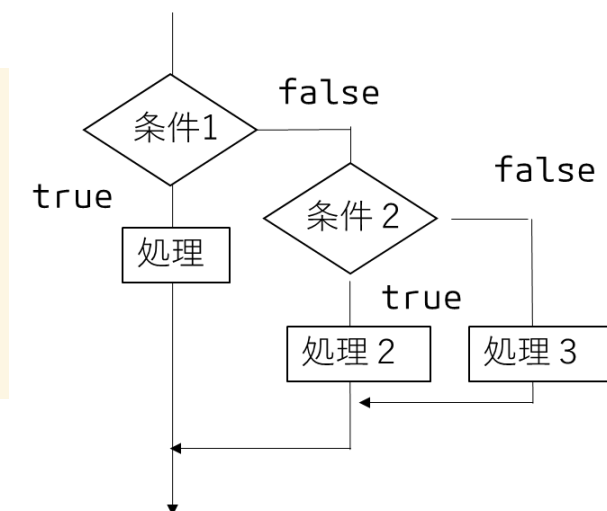
# if-else 文のネスト構造

- if または else の後の中括弧の中の文は一行だけある場合、その中括弧を省略することができます：

```
1 int time = 10;
2 if (time < 18)
3     System.out.println("Good day.");
4 else
5     System.out.println("Good evening.");
```

- これを利用すれば、複数の分岐文を簡潔に連結する（**ネスト**<sup>[Nesting]</sup>）ことができ、分岐が多い状況に対応することができます：

```
1 int time = 10;
2 if (time < 10) System.out.println("Good morning.");
3 else if (time < 18) System.out.println("Good afternoon.");
4 else System.out.println("Good evening.");
```



# switch-case 文

- **switch-case** 文を使用すると、分岐構文を簡単化することができます：

```
switch (expression) {  
    case value1:  
        codes1;  
        break;  
    case value2:  
        codes2;  
        break;  
    default:  
        break;  
}
```

## Note



expression は、整数型  
(long を除く) か文字  
列型しか使えません。

- Java は expression の値に基づいて、case ブロックを選択して実行します。

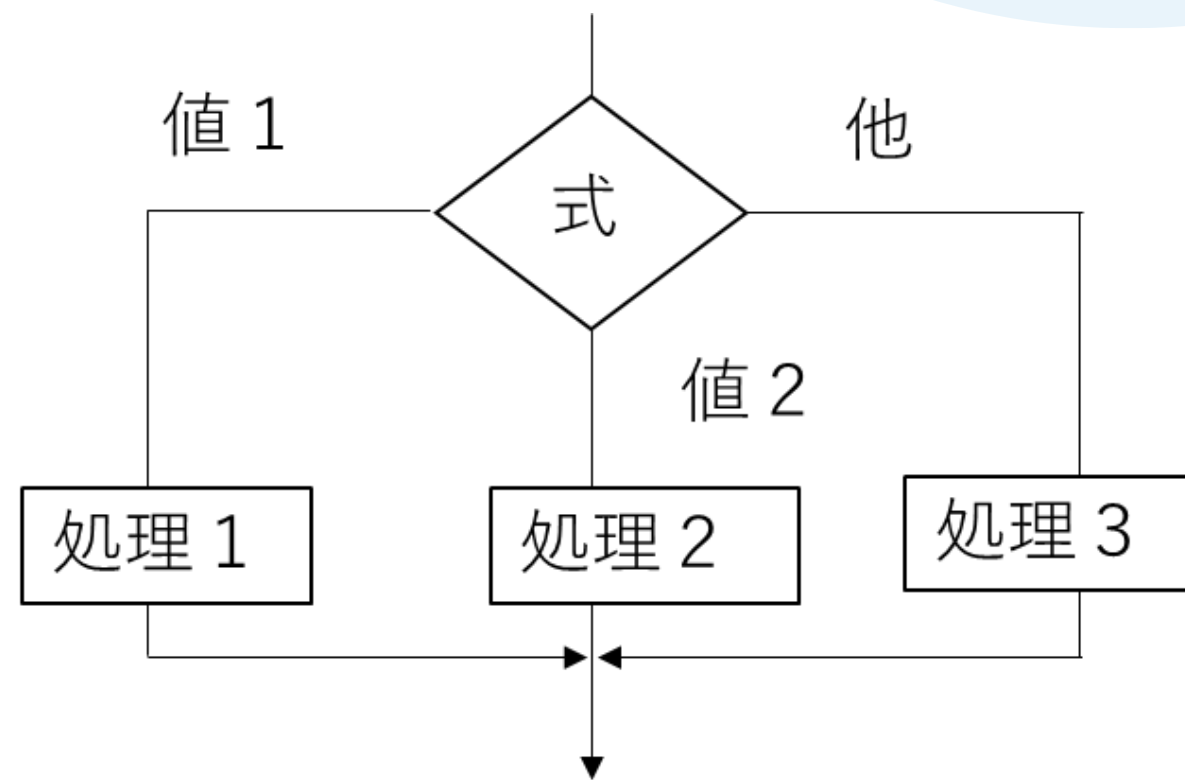
次へ





- **break** キーワードは、switch ブロックから飛び出します。追加されない場合は、選択されたブロック以下のすべての case ブロックも実行されます。
- 一致する case ブロックがない場合、**default** キーワード指定したコードを実行されます。

Try 01011  
Switch.java



# 条件演算子

- **条件演算子**、または**三項演算子**<sup>[Ternary Operator]</sup>は、分岐操作を簡潔に表現するために使用されます：

```
condition ? expression1 : expression2
```

- 条件演算子は if 文と異なり、**値**を直接計算して返します。この値を他の文やメソッドに直接使用することができます。

```
1 int time = 10;
2 System.out.println(time < 18 ? "Good day." : "Good evening.");
```

- if 文を使用する目的が、条件に基づいて特定の値を計算することである場合は、三項演算子の出番になります。

# Q&A

# 目次

- 1 条件分岐
- 2 繰り返し処理
- 3 メソッド

# 繰り返し処理

- 同じコードを複数回繰り返す必要がある場合もあり、そのような場合には**繰り返し文**[Loop Statement]が必要になります。
- Java には、次のような繰り返し文の種類があります：
  - for
  - for-each
  - while
  - do-while



# for 文

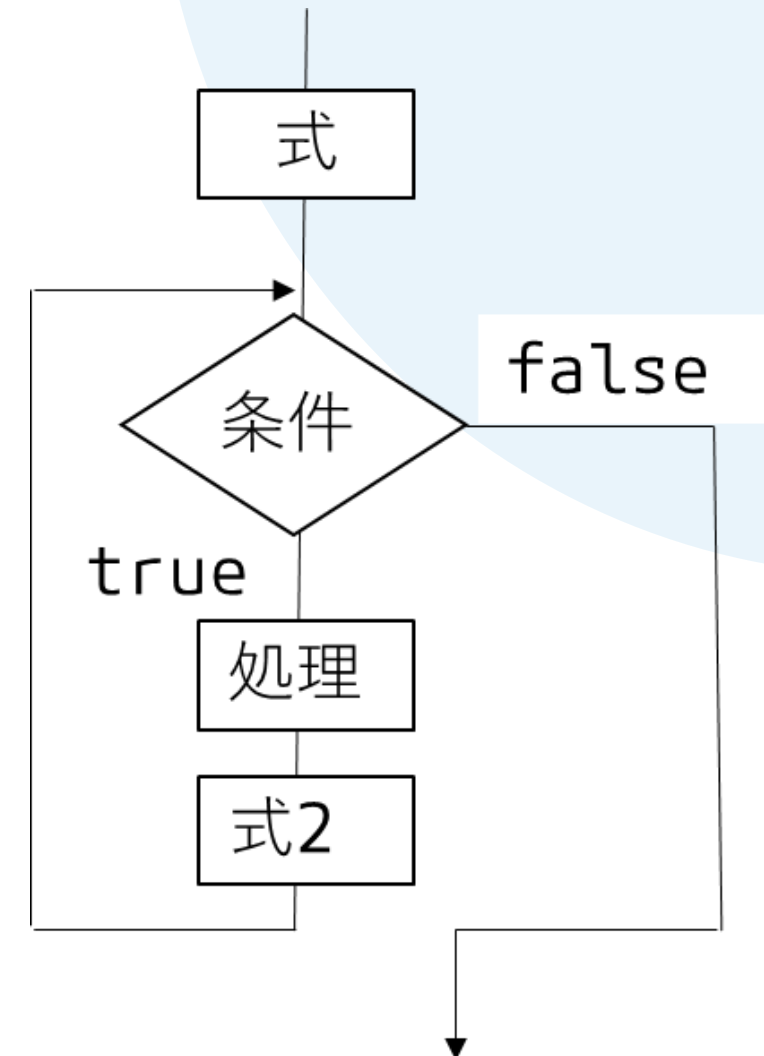
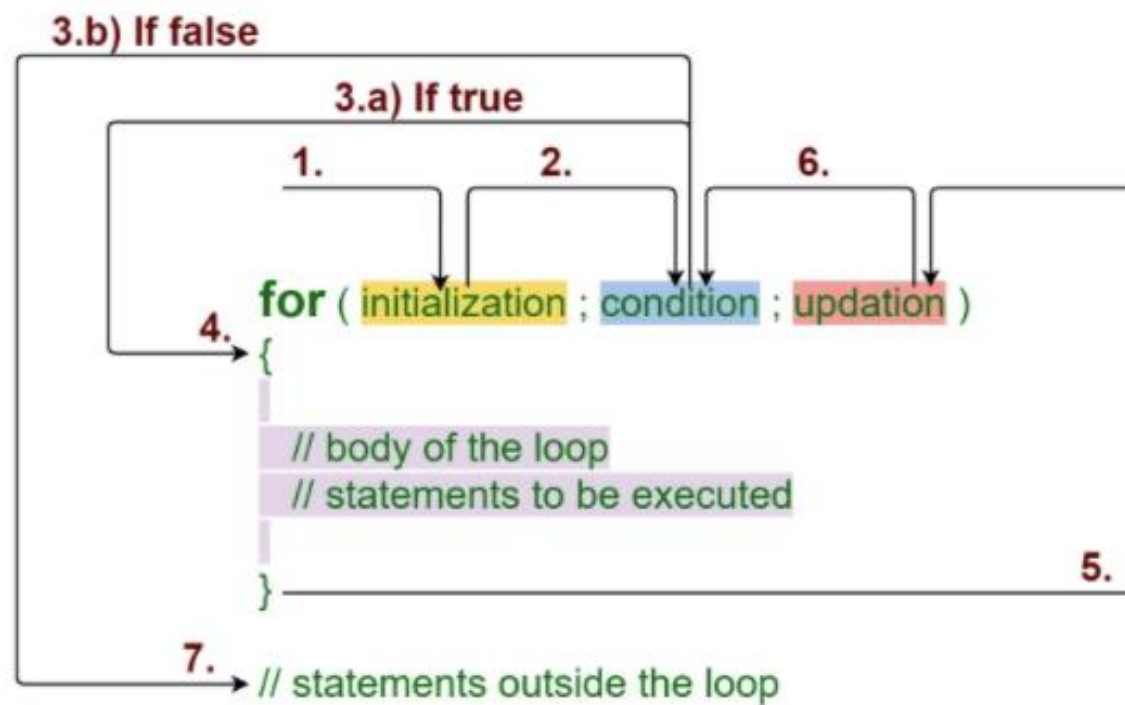
- 例 :

```
for (code init; condition; code after) {  
    code body;  
}
```
- ここで :
  - code init は for 文全体の前に**一度だけ**実行されます。
  - condition **毎回の繰り返し前に判断**されます。偽の場合、繰り返しは終了する。
  - 繰り返しが実行されるたびに、**code body の後に code after** が実行されます。
- **for** 文は、このように繰り返しを**特定の回数**で実行するために使用されることが多いです :

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

次へ 

# ● フローチャート :



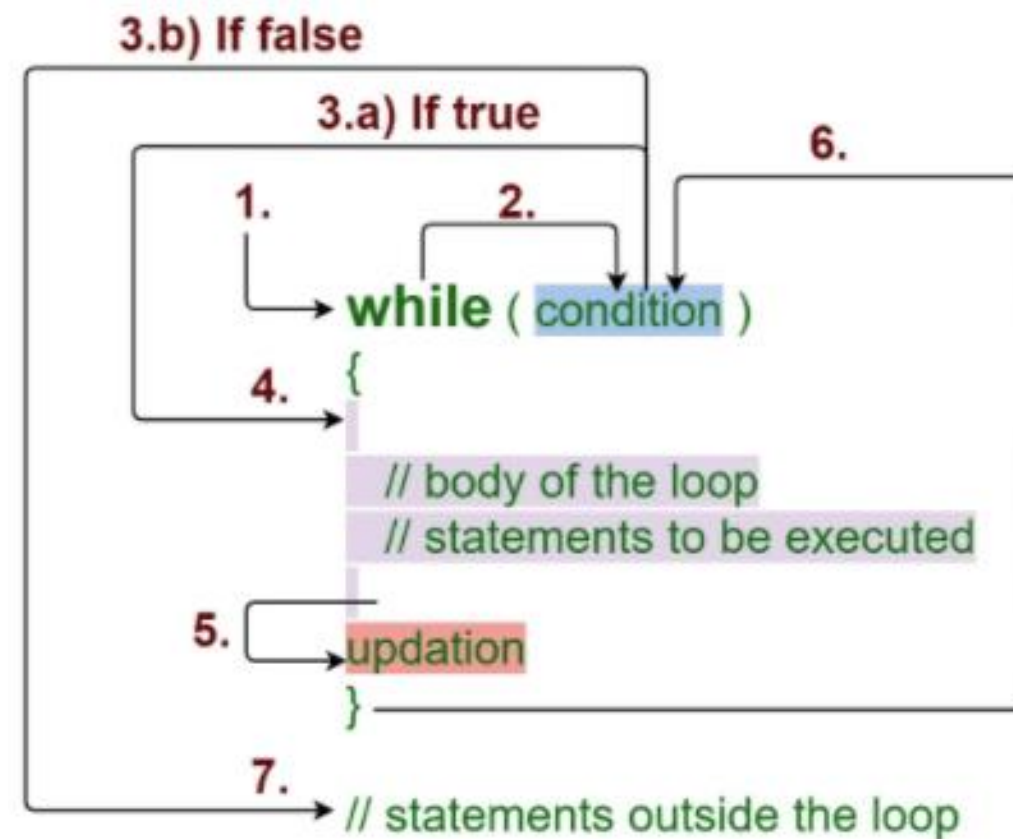
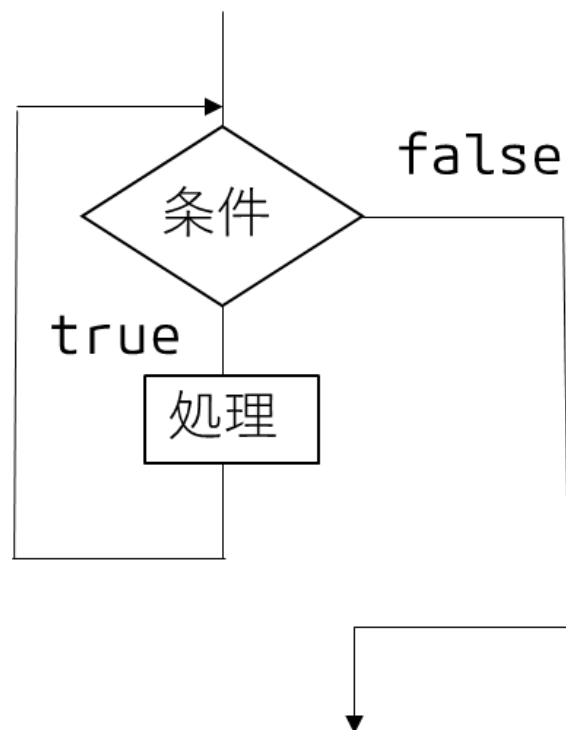
Try  
For.java

# while 文

- **while** 文は、条件を満たしたときにコードのブロックを繰り返し実行します：

```
while (condition) {
    // code
}
```

- フローチャート：



## Try While.java

- 考えてみよう：次のコードはどうなるのでしょうか？

```
1 int i = 10;
2 while (i > 0) {
3     System.out.println(i);
4     i++;
5 }
```

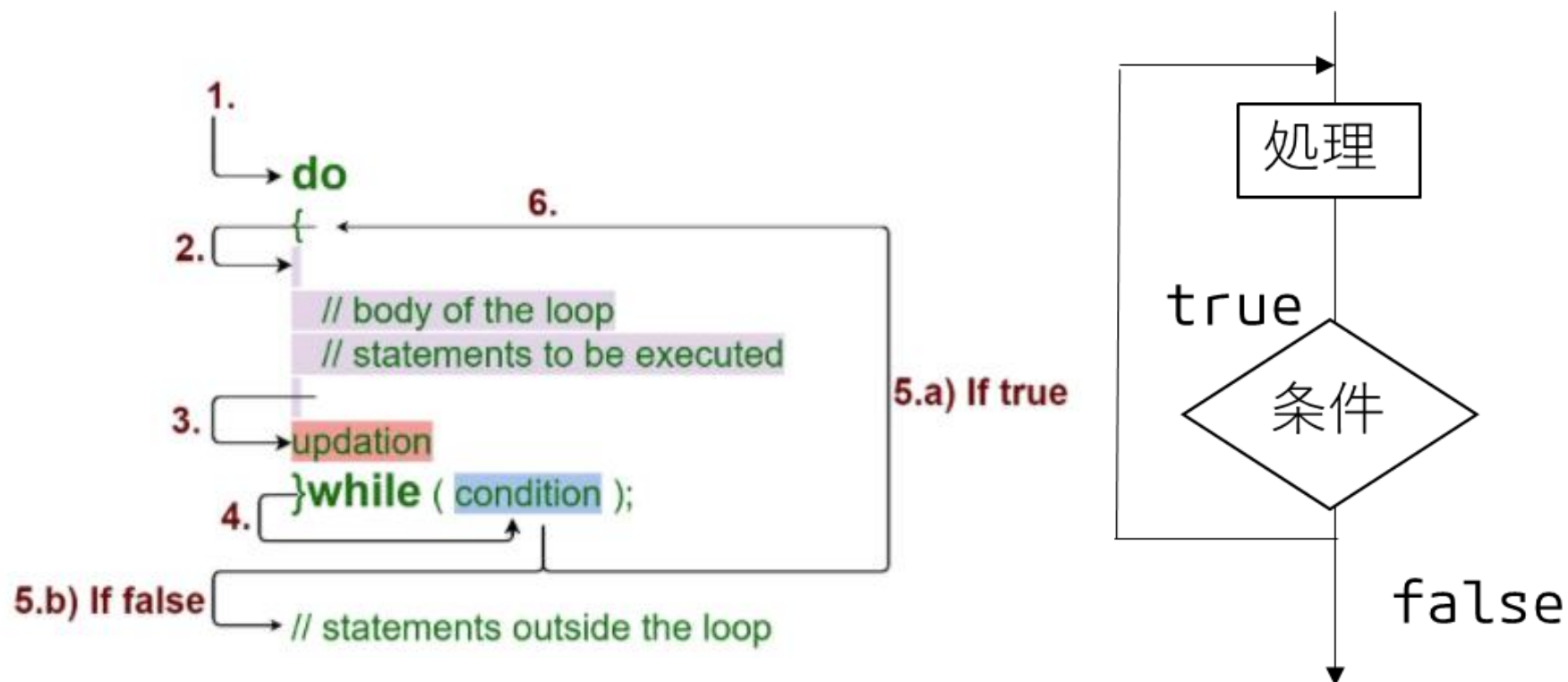
- 条件がずっと満たされていれば、while 文は永遠に止まらない！

# do-while 文

- **do-while** 文は while 文の変形で、条件にかかわらず、繰り返し処理は最低限 1 回実行ます：

```
do {
    // code
} while (condition);
```

- フローチャート：



Try  
DoWhile.java




# for-each 文

- **for-each** 文は、複数のデータを含む**配列やリスト構造**を繰り返し処理に専用の文です :

```
1 for (type variableName : arrayName) {
2     // code
3 }
```

- 例えば、以下のコードでは配列の中の文字列を一つずつ出力します :

```
1 String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
2 for (String i : cars) {
3     System.out.println(i);
4 }
```

Try  ForEach.java

# 練習タイム

- for、while、do-while 繰り返し文を使って、以下の配列を出力してください。

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

- 考えてみよう：上記の各文法は、それぞれどのような場合に使うべきですか？ 具体的な例を挙げて説明してください。

# break 文

- **break** 文は switch 文を抜けるだけでなく、任意の繰り返し文を抜けるにも使用できます：

```
1 for(int i = 1; i<10; i++) {  
2     if(i ==3) {  
3         break;  
4     }  
5     System.out.println(i);// =>1 2  
6 }  
7
```

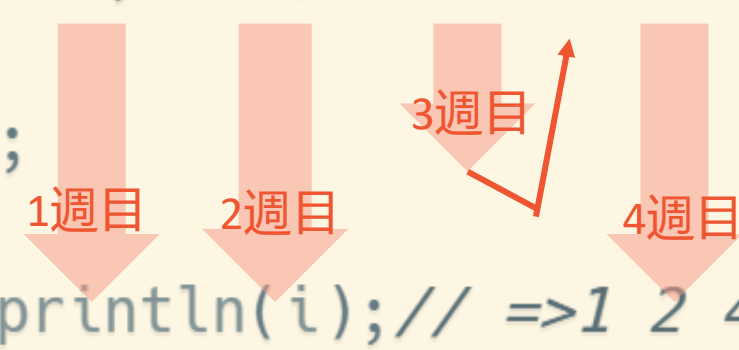
# continue 文

- **continue** 文は、**1 回だけ**の繰り返しを抜けます。

```

1 for(int i = 1; i<5; i++) {
2     if(i ==3) {
3         continue;
4     }
5     System.out.println(i); // => 1 2 4
6 }

```



## Note



continue 文は **1 回の繰り返し**をスキップし、ループ条件を評価し始めますが、break 文は残りの**すべての繰り返し**を直ちに終了します。

# 練習

- 次の2つのコードは、それぞれ何を出力するでしょうか？

```
1 int i = 0
2 while (i < 5) {
3     System.out.println(i);
4     i++;
5     if (i == 2) {
6         break;
7     }
8 }
```

```
1 int i = 0
2 while (i < 5) {
3     if (i == 2) {
4         continue;
5     }
6     i++;
7     System.out.println(i);
8 }
```



# Q&A

## 繰り返しのネストとラベル

多次元配列のような複雑なデータ構造にアクセスする場合、繰り返し文の内に他の繰り返し文を書く必要がある場合があります。例えば、2次元の配列にアクセスする場合、最初のループ（外側ループ）はすべての配列の繰り返し処理を担当し、2番目のループ（内側ループ）はそれらの配列の要素の繰り返し処理を担当します。これを繰り返しのネストと言います。

しかし、break 文も continue 文も、最も近いループ（内側のループ）だけを抜けます。外側のループを抜きたい場合は、**ラベル**を使います。

Try   
Label.java

# 目次

- 1 条件分岐
- 2 繰り返し処理
- 3 メソッド

# メソッド

- 考えてみよう：これまでのプログラミングで、同じような、似たようなコードを何度も書いたことはありますか？
- 同じような機能を何度も使うプログラムであれば、コードを貼り付けることで、開発時間を短縮することができるのでしよう。しかし、これでは読みにくい冗長なコードになりがちです。
- これを解決するためには、**メソッド**<sup>[Method]</sup>を使います。メソッドはコードのブロックを定義し、それを**呼び出す**<sup>[Calling]</sup>ことで同じ機能を実現し、再利用することができます。
- メソッドは、直接コードを貼り付けることよりも柔軟性があります。異なる**引数**を渡すことで、機能を細かく細かく調整して様々な状況に対応することができます。

## Example



「 $2^3 + 3^4 + 8^5$ 」の結果を計算するプログラムを作成したいです。

確かに、次のように書くこともできますが：

```
1 public static void main(String[] args) {  
2     int a = 2 * 2 * 2 + 3 * 3 * 3 * 3 + 8 * 8 * 8 * 8 * 8;  
3     System.out.println(a); // => 32857  
4 }
```

しかし、このプログラムの**可読性**が高いのでしょうか？

それに、このプログラムの**拡張性**はどうでしょうか？

次へ



## Example



もし、「a の b 乗」を計算できるメソッド `power(a, b)` があれば、以下のコードで同じ結果を計算することができます。

```
1 public static void main(String[] args) {  
2     int a = power(2, 3) + power(3, 4) + power(5, 8);  
3     System.out.println(a);  
4 }
```

このプログラムの可読性や拡張性はどうでしょうか？

# メソッドの定義

- Java にはすでに多くの便利なメソッドが用意されており、そのうちのいくつかは以前にも使ったことがあります。それでも、実際の開発では**新しいメソッドを定義する**必要があります。
- メソッドを定義するための基本的な構文は次のとおりです。

```
1 static int power(int a, int b) {
2     codes;
3 }
```

- ここで、power はメソッド名、int は、メソッドの**戻り値**[Return Value]のデータタイプ、int a, int b はメソッドの**パラメータ**[Parameter]のリスト、codes はメソッドが呼び出されたときに実行されるコードです。

**Note**  **main メソッドの外で定義してください！**

# メソッドのパラメータリスト

- 各メソッドは 1 個、0 個あるいは 2 個以上のパラメータを持つことができます。メソッドを作成する際には、メソッドのすべてのパラメータのタイプと名前を指定する必要があります。複数のパラメータはカンマ「**,**」で区切ります。

## Example



パラメータが 0 個:

```
static int answerToEverything( )
```

パラメータが 1 個:

```
static int sqrt(int a)
```

パラメータが 2 個:

```
static int power(int a, int b)
```

# メソッドの戻り値

- メソッドの戻り値は、メソッド内で計算された結果を呼び出された場所に返すために使用されます。メソッドの戻り値は、定義時に明示する必要があります。
- メソッド内で、**return** 文を使ってデータを戻り値として返します（**これでメソッドも即終了！**）：

```
1 static int answerToEverything() {
2     return 42;
3 }
```

- メソッドに戻り値が必要ない場合は、戻り値を「**void**」と定義すればいいです。

```
1 static void sayHello() {
2     System.out.println("Hello!");
3 }
```

# メソッドの呼び出し

- メソッドを定義した後は、プログラムのどこでもメソッドを**呼び出して**、定義したコードを実行することができます。メソッドを呼び出すには、メソッド名の後に「**()**」を書き、各引数の値をカンマで区切って順番にいれます。

```
1 public static void main() {
2     sayHello();           // 引数なしメソッドの呼び出し
3     sqrt(4);              // 引数1つのメソッド
4     System.out.println(power(2, 3)); // 引数2つのメソッド
5 }
```

- なお、このメソッドの戻り値を直接に他のコマンドや操作に利用させることもできます。

Try  Method.java



# メソッドのまとめ

```

1
2 public class Main {
3     public static void main(String[] args) {
4
5         //circleに10という値をボタンで渡して返っていたボタンに
6         //とcircleAreaResultと名付ける
7         double circleAreaResult = circle(10);
8             ⑤                ①
9         System.out.println(circleAreaResult);
10
11     }
12     //渡されたボタンにhankeiと名付ける
13     public static double circle (int hankei) {
14         //hankeiを利用した計算結果の値にcircleAreaと名付ける
15         ③ double circleArea = 3.14 * hankei*hankei;
16         //circleAreaというボタンを外に渡してあげる。
17         ④ return circleArea;
18     }
19 }

```

メソッド名  
↓

パラメータ  
↓ ②

戻り値  
↑

1 : circle()に10というボタン  
(値) を渡してあげる

2 : 渡されたボタンに勝手に  
hankeiと名付けてあげる

3 : ボタンを使った計算結果の値  
をcircleAreaと名付けて新たにバ  
トンを作る

4 : returnでcircleAreaという  
ボタンを外に渡す

5 : 返ってきたボタンに  
circleAreaと名付けてあげる

# 練習タイム

- 整数のパリティを確定するメソッドを作ってください。
- メソッド名 : `checkParity`。
- 引数 : タイプ : `int`、名前 : `num`。
- 戻り値 : タイプ : `String`。
  - 奇数の場合は `"odd"`、偶数の場合は `"even"` を返します。
- `main` メソッドで呼び出され、要件を満たしているかどうかをチェックせよ。



# Q&A

# メソッドのオーバーロード

- Java では、メソッドのパラメータリストのタイプが異なることが保証されている限り、**同じ名前のメソッド**を複数作成することが可能です。これをメソッドの**オーバーロード** [Overload] といいます。

```
static int power(int a, int b);  
static float power(float a, float b);  
static int power(int a);
```

Try   
Overload.java

次へ

- オーバーロードは、機能は似ているが引数の種類が異なるメソッドを統一することができるほか、「デフォルト引数」の実装にも利用することができます：

```
1 static int power(int a) {  
2     return power(a, 2);  
3 }
```

### Note



オーバーロードに使われるメソッドは、異なるタイプのパラメータを持つ必要があります。戻り値だけが異なるメソッドではオーバーロードに使うことができません。

# ブロック

- **ブロック**<sup>[Block]</sup>とは、中括弧「**{ }**」で囲まれたコードの集合のことです。ブロック内で宣言された変数のスコープは、そのブロックの内部です。
- 例外として、for (for-each) 文の括弧「**()**」内で宣言された変数のスコープは、その for 文のブロックの中です。
- なお、制御文を使用せずに、ブロックを直接使用することも可能です。



# ローカル変数

- メソッド内部で宣言された変数のスコープは、メソッド内部となります。そのため、他の方法でアクセスすることはできません。これらは**ローカル変数**[Local Variable]と呼ばれます：

```

1 static int answerToEverything() {
2     // ここでは answer が使えない
3
4     int answer = 42;
5
6     // answer が使える
7     return answer;
8 }
9
10 public static void main() {
11     // ここでは answer が使えない
12     // answer = 0; // => エラー
13 }

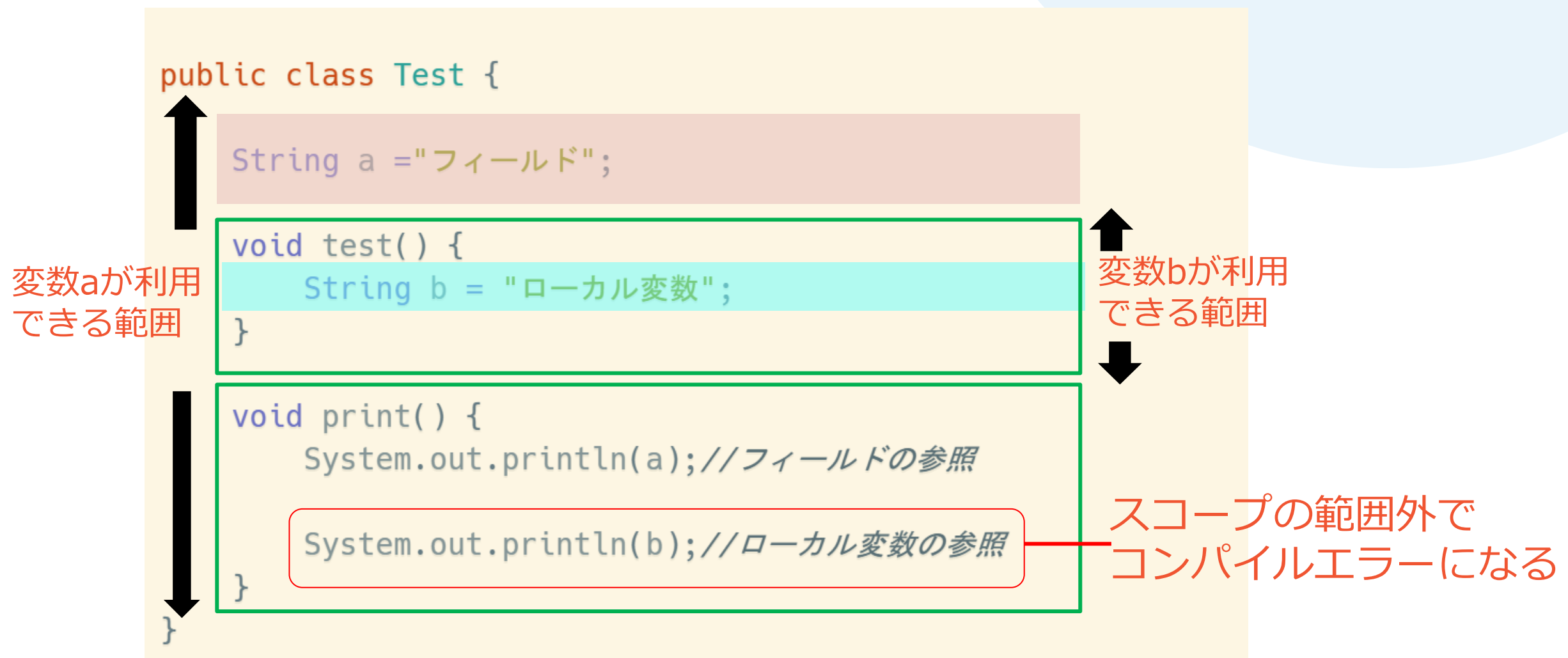
```

## Tips

メソッドのパラメータもそのメソッドの**ローカル変数**になります。

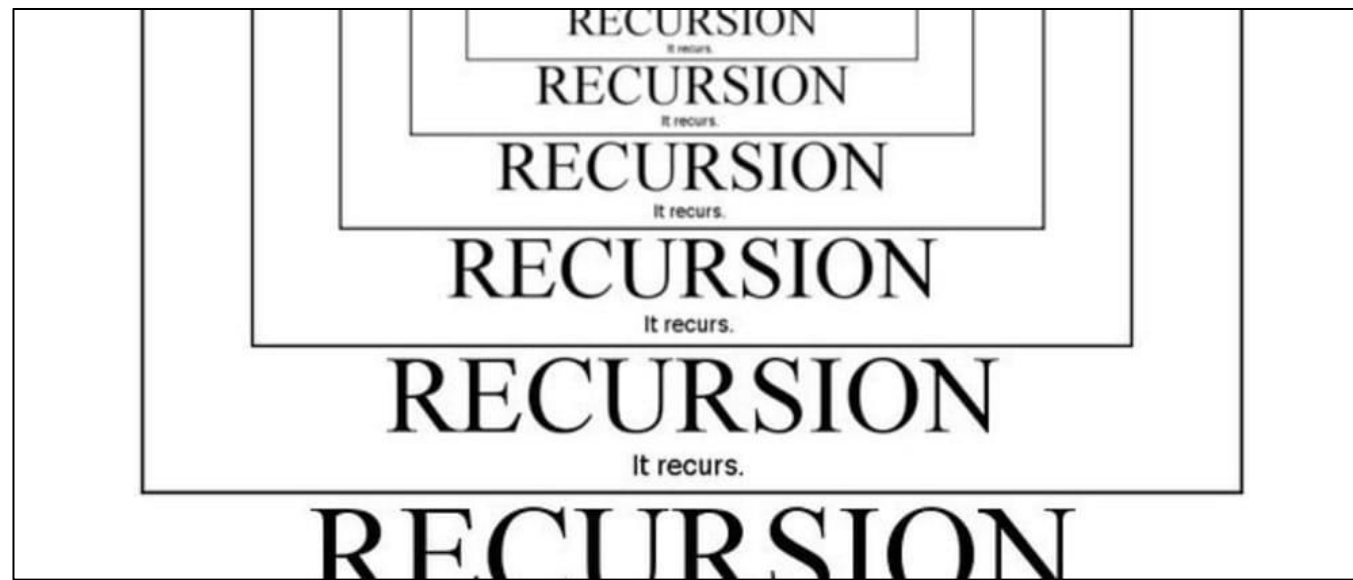
# スコープ

- Java では、変数は**作成された領域内**でのみアクセスすることができます。これを変数の**スコープ**<sup>[Scope]</sup>と呼ばれます。



# メソッドの再帰

- **再帰**[Recursion]とは、メソッドの中で**そのメソッド自身**を呼び出すことです。



- もし、ある問題を、同じ種類のいくつかの小さな問題（部分問題）に分解することができて、かつそれらの部分問題の解を使って元の問題を解くことができるならば、再帰の出番となります。



# 再帰の考え方

- n以下の正の整数の総和を計算するプログラムで再帰とは何かを学びましょう。

```

1 public class Reflexive {
2     public static void main(String[] args) {
3         for (int n = 0; n <= 5; ++n) {
4             System.out.println(n + "までの和=" + f(n));
5         }
6     }
7 }
8 static int f(int n) {
9     if (n == 0) {
10         return 0;
11     } else {
12         return n + f(n-1);
13     }
14 }
15 }

```

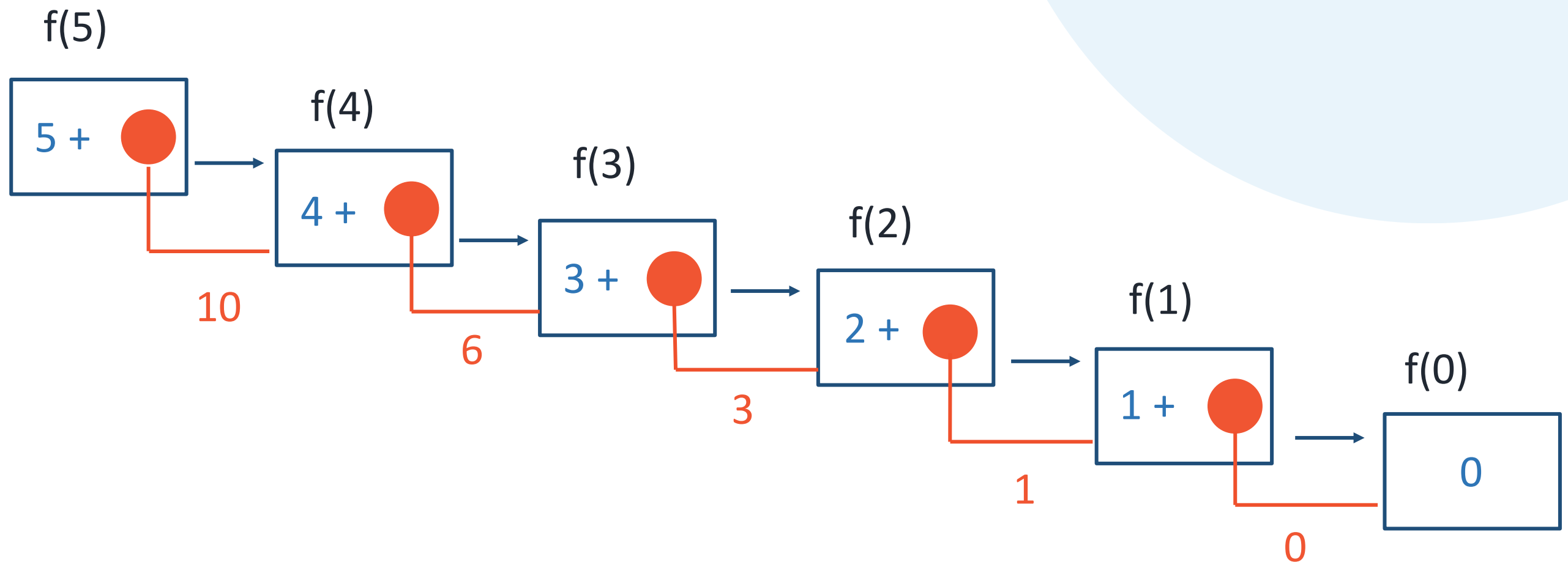


実行結果

0までの和=0  
 1までの和=1  
 2までの和=3  
 3までの和=6  
 4までの和=10  
 5までの和=15

# 再帰の考え方

- 先程のプログラムを図で考えていきましょう。



# 再帰の考え方

- $n = 0$  のとき
  - `if (n == 0) return 0;`
  - によって一瞬で 0 が返って来ます。

- $n = 1$  のとき

```
return n + f(n-1);
```

- $1 + f(0)$  を計算しようとして、 $f(0)$  を呼び出す
- $f(0)$  は、0 を返す
- よって、 $1 + f(0) = 1 + 0 = 1$  となる
- その値を return するという流れになっています。よって 1 が返って来ることになります。

# 再帰の考え方

- $n = 2$  のとき
  - $2 + \text{func}(1)$  によって  $\text{func}(1)$  が呼び出される
  - $\text{func}(1)$  の中では  $1 + \text{func}(0)$  によって、 $\text{func}(0)$  が呼び出される
  - $\text{func}(0)$  では  $0$  を返す
  - よって、 $\text{func}(1)$  が  $1 + 0$  となり
  - 更に  $2 + \text{func}(1)$  の  $\text{func}(1)$  に  $1 + 0$  代入すると  $2 + 1 + 0$  となり、
  - 結果、 $3$  が返って来ます。

# 再帰の終了条件と書き方

- メソッドに再帰的な呼び出しをさせるわけにはいきません。再帰的なメソッドを書くときには、そのメソッドに何らかの**終了条件**<sup>[Halting Condition]</sup>の存在を保証することが重要です。つまり、問題が十分に小さいとき、再帰的な呼び出さずに、直接答えを見つけます。

```

アクセス修飾子 戻り値の型 メソッド名(引数) {
    if (終了条件) {
        return 戻り値;
    } else {
        メソッド名(引数);
        return 戻り値;
    }
}

```

Try  Recursion.java

# Q&A

# まとめ

Sum Up



1.条件分岐 : **if-else**、switch-case、 ? :。

2.繰り返し処理 :

- ① **for**、 **while**、 do-while、 for-each
- ② break と continue

3.メソッド :

- ① 宣言と使用
- ② オーバロード
- ③ スコープ
- ④ 再帰





*Light in Your Career.*  
**THANK YOU!**