プログラミング入門1

第9回

メソッド（3）

授業の前に自己点検
以下の質問に答えられますか？

- メソッドの宣言とは、起動とは何ですか
- メソッドの宣言はどのように書きますか
- メソッドの宣言はどこに置きますか
- メソッドの起動はどのようにしますか
- メソッドの仮引数、実引数、戻り値とは何ですか
- メソッドの起動にあたって実引数はどのようにして仮引数に渡されますか
- 戻り値はどのように利用しますか
- 変数のスコープとは何ですか

前回の例で復習

```
public class Combination {
  public static void main(String[] args) {
    //combinationメソッドを呼び出す
    System.out.println("10人から2人選ぶ組み合わせは"
      + combination(10, 2) + "通り");
  }
  // nCr = n! / ((n-r)! * r!)を計算するメソッド
  public static int combination(int n, int r) {
    // 階乗の計算でfactメソッドを呼び出す
    return fact(n) / (fact(n-r) * fact(r));
  }
  //nの階乗を計算するメソッド
  public static int fact(int n) {
    int total = 1;
    for (int i = n; i >= 1; i--) {
      total *= i;
    }
    return total;
  }
}
```

メソッドの宣言を置く場所
クラスブロック内、メソッド同士(mainメソッドも)は入れ子にならない

```
public class Combination {

        mainメソッドの前でもよい

    public static void main(String[] args) {
        //combinationメソッドを呼び出す
        System.out.println("10人から2人選ぶ組み合わせは"
            + combination(10, 2) + "通り");
    }

        mainメソッドの後でもよい

}
```

メソッドの宣言の書き方

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

    System.out.println("10人から2人選ぶ組み合わせは"
      + combination(10, 2) + "通り");
  }

  public static int combination(int n, int r) {
    // 階乗の計算でfactメソッドを呼び出す
    return fact(n) / (fact(n-r) * fact(r));
  }

  public static int fact(int n) {
    int total = 1;
    for (int i = n; i >= 1; i--) {
      total *= i;
    }
    return total;
  }
}
```

仮引数の宣言

戻り値の型

値を返す命令

メソッドの起動：戻り値があるときは式の中で起動できる

```
public class Combination {
  public static void main(String[] args) {
    System.out.println("10人から2人選ぶ組み合わせは"
      + combination(10, 2) + "通り");
  }

  public static int combination(int n, int r) {
    // 階乗の計算でfactメソッドを呼び出す
    return fact(n) / (fact(n-r) * fact(r));
  }
  public static int fact(int n) {
    int total = 1;
    for (int i = n; i >= 1; i--) {
      total *= i;
    }
    return total;
  }
}
```

println命令の引数になる式

int型の戻り値は文字列に変換されて、式の一部になる

メソッドの起動：戻り値があるときは式の中で起動できる

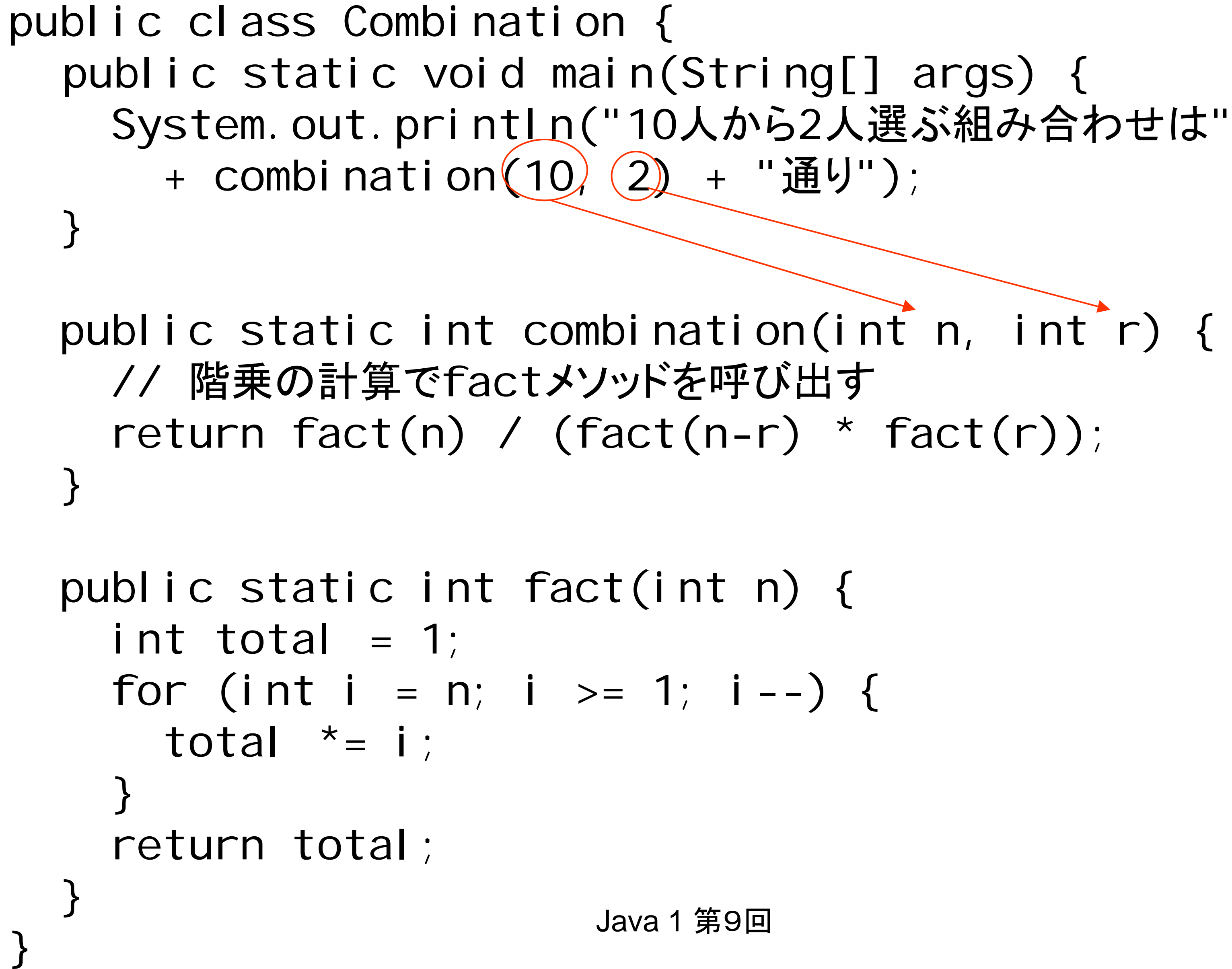
```
public class Combination {
  public static void main(String[] args) {
    System.out.println("10人から2人選ぶ組み合わせは"
      + combination(10, 2) + "通り");
  }

  public static int combination(int n, int r) {
    // 階乗の計算でfactメソッドを呼び出す
    return fact(n) / (fact(n-r) * fact(r));
  }
  public static int fact(int n) {
    int total = 1;
    for (int i = n; i >= 1; i--) {
      total *= i;
    }
    return total;
  }
}
```

return文の式

int型の戻り値はreturn文の式の一部になっている

実引数と仮引数の対応



変数のスコープ

```
public class Combination {
  public static void main(String[] args) {
    System.out.println("10人から2人選ぶ組み合わせは"
      + combination(10, 2) + "通り");
  }

  public static int combination(int n, int r) {
    // 階乗の計算でfactメソッドを呼び出す
    return fact(n) / (fact(n-r) * fact(r));
  }

  public static int fact(int n) {
    int total = 1;
    for (int i = n; i >= 1; i--) {
      total *= i;
    }
    return total;
  }
}
```

スコープが重ならないから無関係

今回のテーマ

- クラスフィールド（クラス変数）
- スコープ
  - クラスフィールドはどのメソッドからも参照できる変数
  - ローカル変数は宣言されたブロック内
- 寿命
  - 自動変数との違い
  - 今まで使ってきたローカル変数は自動変数
- 再帰関数

クラスフィールドの宣言

```
public class Count {
  static int count;

  public static void main(String[] args) {
    count = 0;
    countUp();
    countUp();
    countUp();
    countDown();
    countDown();
    countDown();
  }

  public static void countUp() {
    count++;
    System.out.println("1増やしたカウンタの値は" + count);
  }

  public static void countDown() {
    count--;
    System.out.println("1減らしたカウンタの値は" + count);
  }
}
```

staticキーワードが必要

すべてのメソッドの外で宣言

クラスフィールドのスコープ

```java
public class Count {
  static int count;

  public static void main(String[] args) {
    count = 0;
    countUp();
    countUp();
    countUp();
    countDown();
    countDown();
    countDown();
  }

  public static void countUp() {
    count++;
    System.out.println("1増やしたカウンタの値は" + count);
  }

  public static void countDown() {
    count--;
    System.out.println("1減らしたカウンタの値は" + count);
  }
}
```

参照や値の変更が可能

実行すると

```
public class Count{
  static int count;

  public static void main(...){
    count = 0;
    countUp();
    countUp();
    countUp();
    countDown();
    countDown();
    countDown();
  }
  ...
}
```

```
1増やしたカウンタの値は1
1増やしたカウンタの値は2
1増やしたカウンタの値は3
1減らしたカウンタの値は2
1減らしたカウンタの値は1
1減らしたカウンタの値は0
```

ローカル変数、クラスフィールドのスコープ

- メソッドの中で宣言 (定義ともいう) されている変数 (ローカル変数と呼ぶ) は、それを宣言したブロックの内部でのみ有効。
- ブロックとはプログラムの中で{…}で示される範囲を言う。ブロックは幾重にもネスティング(入れ子の形になる)されることがある。
- あるブロックで定義された変数は、その内部のブロックに同じ名前の定義がない限りこの内部で有効。同じ名前が内部にあるとその内部では外側の名前は使えない。
- クラスフィールドはどのメソッドからも参照、値の変更が可能。

ローカル変数、クラスフィールドの寿命

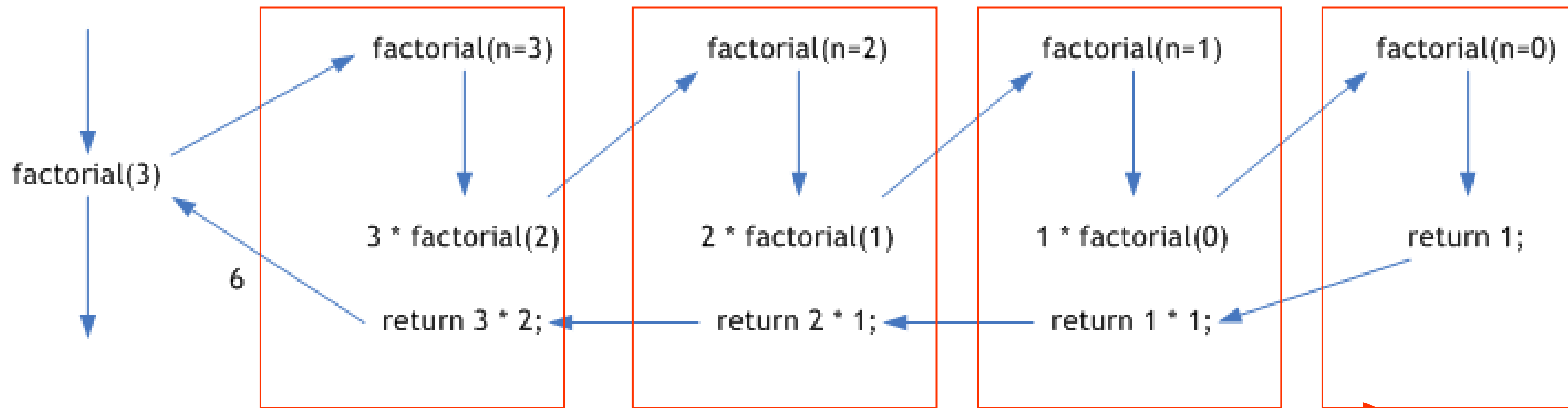
- ローカル変数は、それを定義しているブロックを実行し始めたとき用意され（メモリ上にその箱が作られ）、ブロックを終了したとき消失する。
- このような変数を自動変数という。
  - 必要なときに（ブロックに入ったときに）自動的に作られ、必要がなくなったときに（ブロックをぬけたときに）自動的に消失するという意味。
- クラスフィールドはローカル変数と違って、消失することは無い。最後に書き込んだ値を参照することができる。

階乗を再帰的(recursive)に定義する

考え方
factorial(N) = 1　　N = 0 のとき
factorial(N) = N * factorial(N-1)　それ以外のとき

```
public static int factorial(int n) {
  // factorial(N) = 1 (N = 0 のとき)
  if (n == 0) {
    return 1;
  }
  // factorial(n) = n * factorial(n-1) (それ以外のとき)
  else {
    return n * factorial(n - 1);
  }
}
```

factorial(3) として起動すると



factorial (0) が実行されているときは4つの実行中あるいは再帰呼び出しからの帰り待ちの状態のメソッド起動がある。

factorial メソッドの仮引数である自動変数nはfactorial の呼び出しの度に独立して自動的に作られ、消されるので、ソースプログラム上で同じ変数nであっても実行中の異なるメソッド起動では実体が異なる。

行きがけの処理
課題0804の再帰的な書き換え

```
public class PrimeFactorsRecursiveLeft {
  public static void main(String[] args) {
    printLeft(210);
  }
  public static void printLeft(int n) {
    if (n == 1)
      return;
    int mpf = minimumPrimeFactorOf(n);
    System.out.print(mpf + " ");
    printLeft(n / mpf);
  }
  public static int minimumPrimeFactorOf(int n) {
    if (n == 1)
      return 1;
    else
      for (int i = 2; i <= n / 2; i++)
        if (n % i == 0)
          return i;
    return n;
  }
}
```

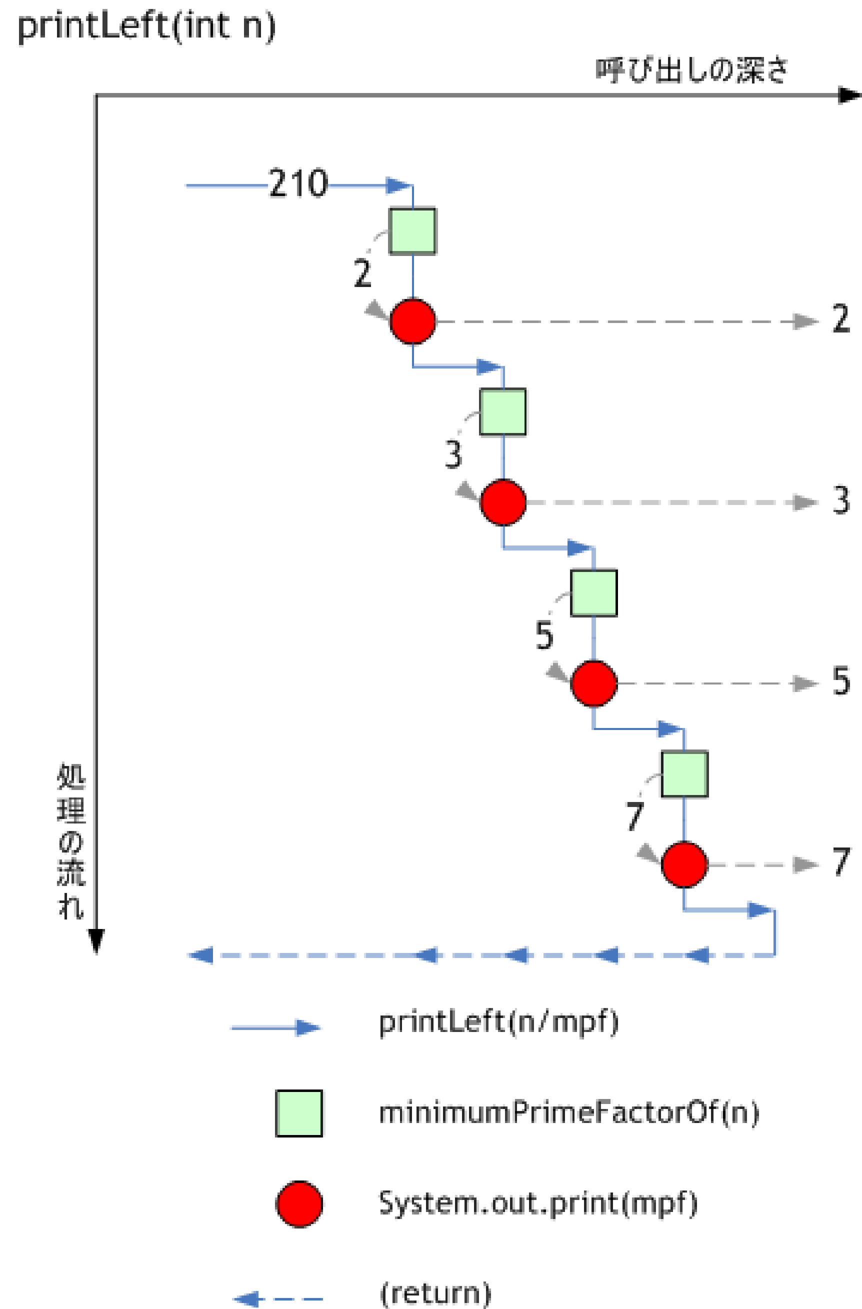
行きがけの処理

実行結果：２３５７

```
public class PrimeFactorsRecursiveLeft {
  public static void main(String[] args) {
    printLeft(210);
  }
  public static void printLeft(int n) {
    if (n == 1)
      return;
    int mpf = minimumPrimeFactorOf(n);
    System.out.print(mpf + " ");
    printLeft(n / mpf);
  }
  ...
}
```

再帰呼び出し

行きがけの処理



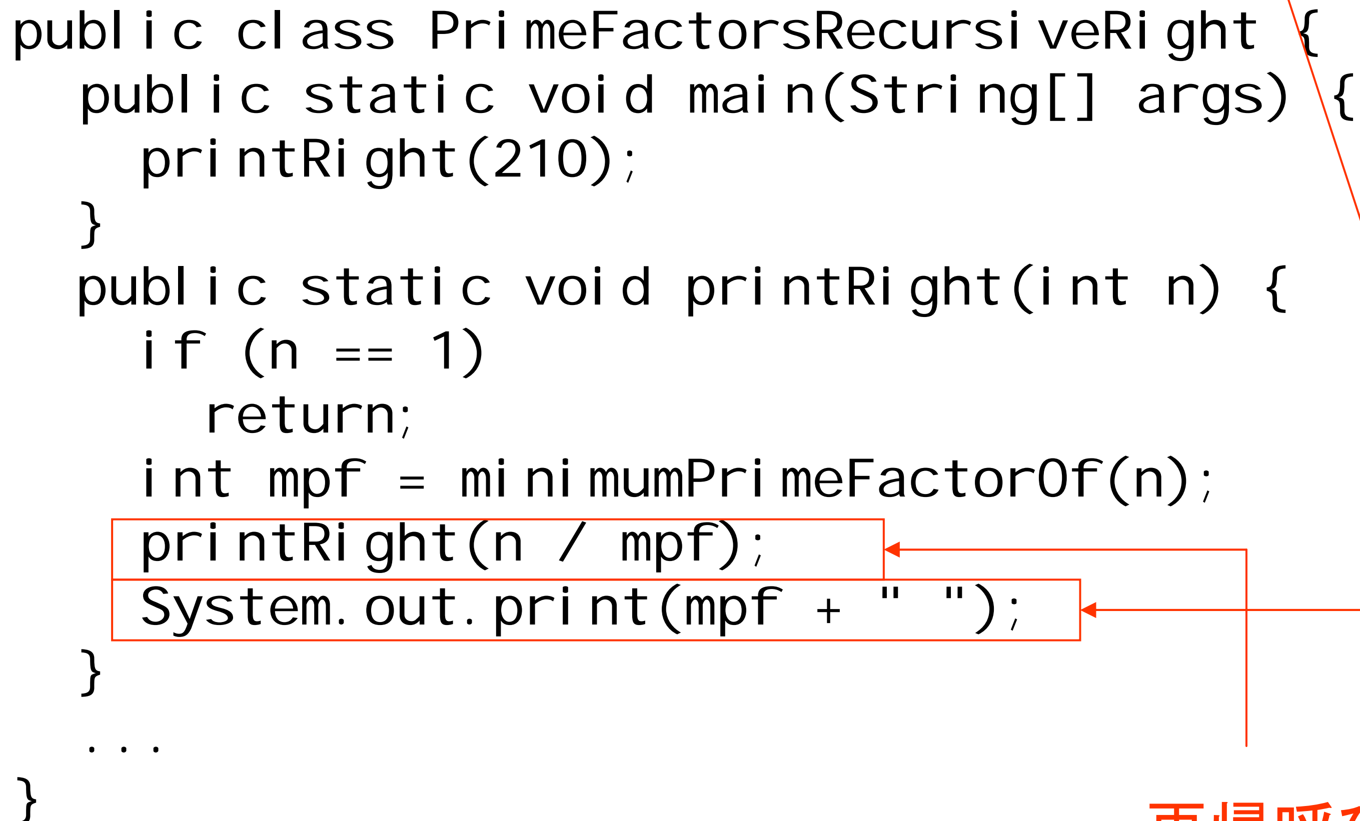
帰りがけの処理

課題0804のもうひとつの再帰的な書き換え

```
public class PrimeFactorsRecursiveRight {
  public static void main(String[] args) {
    printRight(210);
  }
  public static void printRight(int n) {
    if (n == 1)
      return;
    int mpf = minimumPrimeFactorOf(n);
    printRight(n / mpf);
    System.out.print(mpf + " ");
  }
  public static int minimumPrimeFactorOf(int n) {
    if (n == 1)
      return 1;
    else
      for (int i = 2; i <= n / 2; i++)
        if (n % i == 0)
          return i;
    return n;
  }
}
```

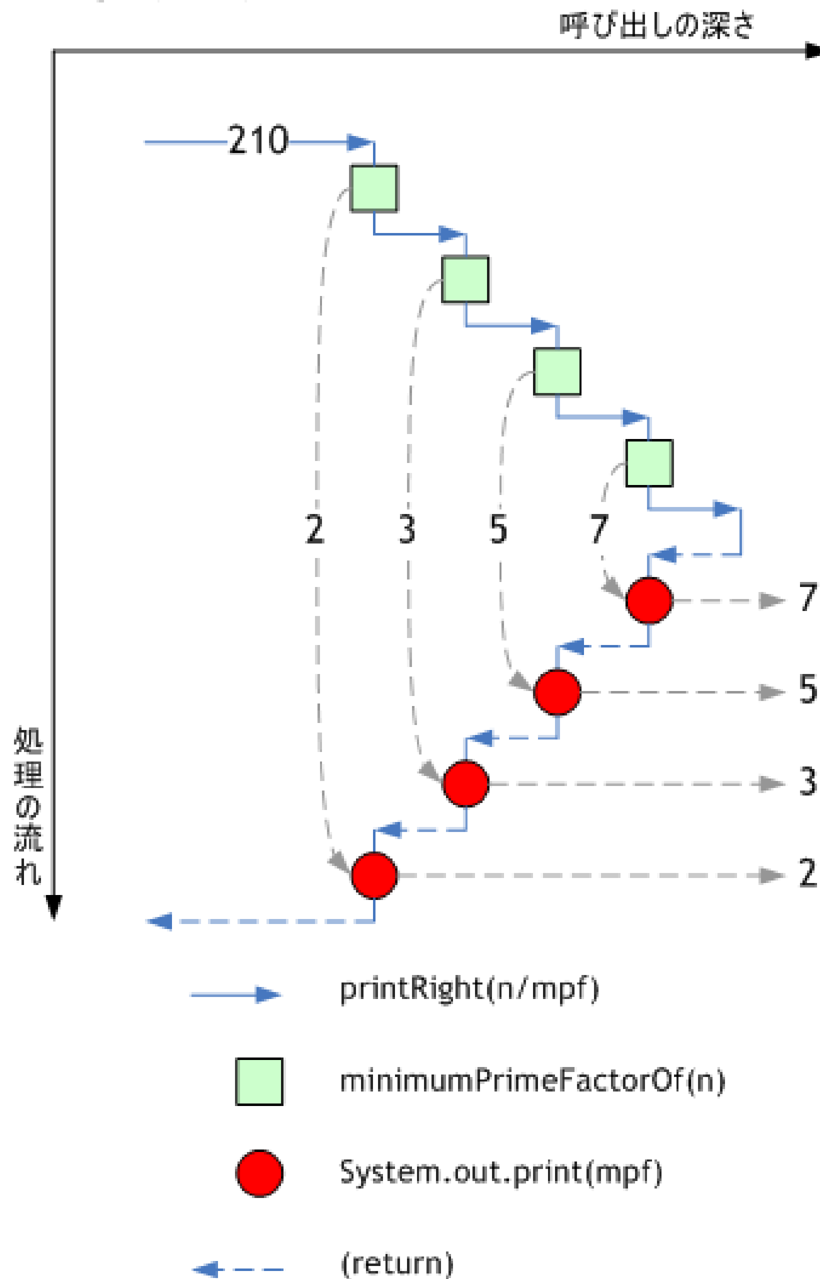
帰りがけの処理

実行結果: 7 5 3 2



```
public class PrimeFactorsRecursiveRight {
  public static void main(String[] args) {
    printRight(210);
  }
  public static void printRight(int n) {
    if (n == 1)
      return;
    int mpf = minimumPrimeFactorOf(n);
    printRight(n / mpf);
    System.out.print(mpf + " ");
  }
  ...
}
```

帰りがけの処理



行きがけの処理と帰りがけの処理

- printLeftとprintRightでは自分自身を再帰的に呼び出すタイミングが異なる。
- printLeftは先に表示してから自分自身を呼び出すのに対し、printRightは自分自身を呼び出してから後に表示を行っている。
- この差異により、結果が表示される順番が変わってくる。
- 再帰メソッドは呼び出すタイミングを調整するだけで、処理の流れを大きく変えることができる。

非再帰化と効率

- 一般に、メソッドの起動は時間や計算機リソースを多く消費する。再帰メソッドはプログラムが簡潔で読みやすいものになっているが、プログラムの効率を考えた場合余り良い選択とはいえない。
- 性能を求められるプログラムを作成する場合は、再帰的な表現をやめて、for 文や while 文を用いて非再帰的な表現に変換することがある。
- 演習で行うフィボナッチ数列のプログラムなどは、非再帰化した表現を探すのは難しい。再帰メソッドの末尾で一度だけ自己呼び出しをするような再帰メソッドは、大抵は for 文などのループ構造で簡単に表現することができる。

非再帰化の例

```
public static int factorial(int n) {
  if (n == 0) {
    return 1;
  }
  int result = 1;
  for (int i = 1; i <= n; i++) {
    result *= i;
  }
  return result;
}
```

一緒にやってみよう

- 今回の演習で使うテストドライバをいつものように指示通り正確にインストールする
  - テストドライバの導入に成功すると
    - プロジェクト「java20XX」の中の「test」というフォルダに「j1.lesson09.xml」という名前のファイルが作成される。
    - このファイルには今週使用するテスト一式が記述されている。
- j1.lesson09 というパッケージを作成する
- 講義資料にあるFactorial, Fibonacciというプログラムを、このパッケージに作成する
  - 講義資料にある手順でテスト、実行までやること

Fibonacciの解説

```
public class Fibonacci {
  static int count;

  public static void main(String[] args) {
    for (int i = 0; i <= 10; i++) {
      count = 0;
      System.out.println("fibonacci(" + i + ") = " + fibonacci(i));
      System.out.println("起動回数は" + count + "回");
    }
  }
  // フィボナッチ数列の 第 k 項を計算するメソッド
  public static int fibonacci(int k) {
    count++;
    if (k == 0) return 0;
    else if (k == 1) return 1;
    else
      return fibonacci(k - 1) + fibonacci(k - 2);
    }
  }
}
```

fibonacci(i)を起動したときに再帰的に呼び出された回数を数えるためのクラスフィールド

```
public class Fibonacci {
  static int count;

  public static void main(String[] args) {
    for (int i = 0; i <= 10; i++) {
      count = 0;
      System.out.println("fibonacci(" + i + ") = " + fibonacci(i));
      System.out.println("起動回数は" + count + "回");
    }
  }
  // フィボナッチ数列の 第 k 項を計算するメソッド
  public static int fibonacci(int k) {
    count++;
    if (k == 0) return 0;
    else if (k == 1) return 1;
    else
      return fibonacci(k - 1) + fibonacci(k - 2);
    }
  }
}
```

クラスフィールドの値を変更

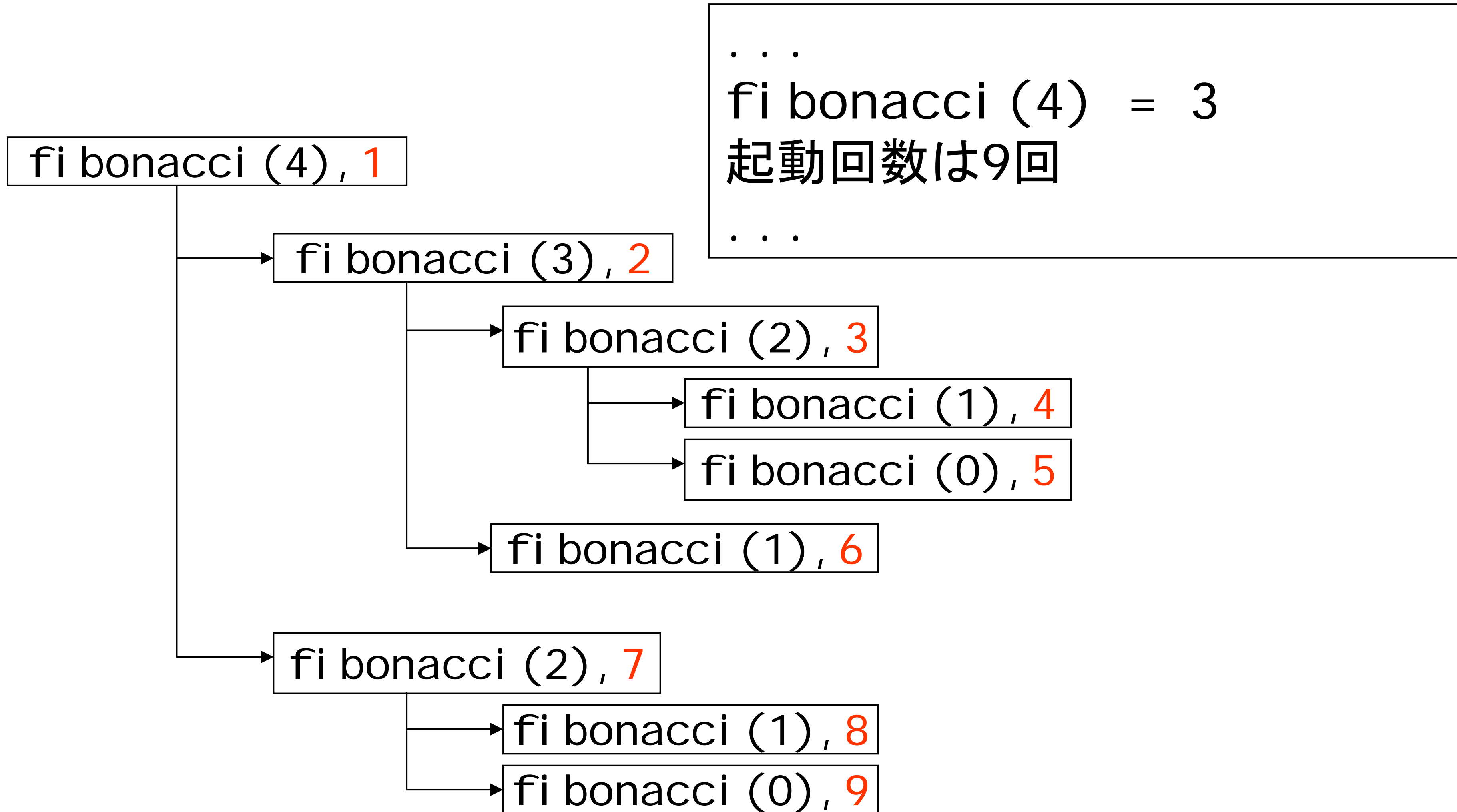
```
public class Fibonacci {
  static int count;

  public static void main(String[] args) {
    for (int i = 0; i <= 10; i++) {
      count = 0;
      System.out.println("fibonacci(" + i + ") = " + fibonacci(i));
      System.out.println("起動回数は" + count + "回");
    }
  }
  // フィボナッチ数列の 第 k 項を計算するメソッド
  public static int fibonacci(int k) {
    count++;
    if (k == 0) return 0;
    else if (k == 1) return 1;
    else
      return fibonacci(k - 1) + fibonacci(k - 2);
  }
}
```

iが3のときの再帰呼び出しの連鎖とクラスフィールドcountの値の変化

```
・・・
fibonacci (3)  =  2
起動回数は5回
・・・
```

iが4のときの再帰呼び出しの連鎖とクラスフィールドcountの値の変化



iが0から10まで走ると

```
fibonacci(0) = 0
起動回数は1回
fibonacci(1) = 1
起動回数は1回
fibonacci(2) = 1
起動回数は3回
fibonacci(3) = 2
起動回数は5回
fibonacci(4) = 3
起動回数は9回

. . .

起動回数は67回
fibonacci(9) = 34
起動回数は109回
fibonacci(10) = 55
起動回数は177回
```

課題

各自のペースで
「第09週目の課題」をやってみよう

課題0902のヒント

```
main
      mとnの入力
      gcd(m,n)の結果を表示



gcd(m,n)
      if n=0
            return m
      else
            return gcd(n,mをnで割った余り)
```

Pascalの三角形

```
main
    n, r への入力の処理
    combinationをn,rを引数として呼び出し、結果を表示する

combination(n,r)
    r=0 または r=n なら 1 を返す
    そうでなければ
        combination(n-1, r-1) と combination(n-1,r) の和を返す
```

Lowest Term

```
main
    分子をnに入力
    分子<1 なら
        警告を出力して return
    分母をdに入力
    分母<1 なら
        警告を出力して return
    printWithReduce をnとdを引数として呼び出す。

printWithReduce(n,d)
    gcd をnとdを引数として呼び出し、戻り値を受け取る
    約分を実行し、結果を表示する

gcd(m,n)
    もし n=0 なら m を返す
    そうでなければ
        gcd(n, m を nで割った余り) を返す
```