プログラミング言語

名倉 正剛
日本大学 工学部 情報工学科
70号館7044号室

授業の日程　※変更する可能性あり

9/17	ガイダンス・復習
9/24	プログラム言語の歴史
10/1	構文，意味，BNF
10/8	識別子，変数
10/15	文，式
10/22	データ型，制御構造
10/29	スコープ，手続き
11/5	中間試験

11/12	モジュール
11/19	データ抽象化
11/26	（月曜授業日）
12/3	（土曜授業日）
12/10	例外処理
12/17	並行，排他制御
12/24	関数型言語，OO言語，言語の分類
1/14	まとめ
1/21	期末試験

プログラミング言語　第7回

- スコープ
- 手続き（途中まで）
- 元のシラバスには，制御構造と書いてあったが，文のところでやってしまった．

- 来週は中間試験．範囲は先週のところまで．
  - 詳しくは先週のスライドを参照してください．

スコープ

- scope (英)：範囲，視野
- ある変数や関数やラベルや型などが，特定の名前で参照される範囲のこと

- ある範囲の外に置いた変数，関数は，通常はその名前だけでは参照できない＝スコープ外である，と言う

分類1:プログラム構文の範囲による分類

- 大域スコープ (global scope)
  - プログラム全体から参照できるスコープ
    - ファイルに関係なく，プログラム全体
- ファイルスコープ (file scope)
  - ファイル内からのみ参照できるスコープ
- 局所スコープ (local scope)
  - 関数内や，ブロック内でのみ参照できるスコープ
- インスタンススコープ (instance scope)
  - オブジェクト指向言語で，メンバ関数のみによって参照できるスコープ (カプセル化)
- クラススコープ (class scope)
  - 特定のクラス内でグローバルに参照できるスコープ

大域スコープ

- （ファイルに関係なく）プログラム全体から参照できる.
- C 言語の場合で言うところの，グローバル変数だけではない

main.c

```
#include <stdio.h>
extern int val;
extern void func(void);

int main(){
  val = 1;
  func();
  printf("main:val=%d\n", val);
  return 0;
}
```

func.c

```
#include <stdio.h>
int val;

void func(){
  printf("func:val=%d\n",val);
  val++;
}
```

```
% gcc –o test main.c b.c
% ./test
```

実行結果

```
#include <stdio.h>
extern int val;
extern void func(void);

int main(){
  val = 1;
  func();
  printf("main:val=%d\n", val);
  return 0;
}
```

```
#include <stdio.h>
int val;

void func(){
  printf("func:val=%d\n",val);
  val++;
}
```

```
% gcc –o test main.c b.c
% ./test
func:val=1
main:val=2
```

- C言語の extern キーワード＝外部定義であることを表す修飾子
  - 関数外で記述すると，ファイル外で定義されていることを示す．

大域スコープ

- 言語によっては，外部に定義されていることを示すキーワードと併せて使う必要がある.

- 言語によっては，大域スコープしか持たない言語もある.（例：BASIC）

ファイルスコープ

- ファイル内であれば，どこからでも参照できるスコープ
- C 言語でグローバル変数を定義すると，通常はこれになる．
  - 右の例で extern の行は省略可能．

```
#include <stdio.h>
int n;

void sub1( void ) {
    extern int n;
    n = 20;
    printf( "n (sub1) = %d\n", n );
}

int main( void ) {
    extern int n;
    n = 10;
    printf( "n (main) = %d\n", n );
    sub1();
    printf( "n (main) = %d\n", n );
    return 0;
}
```

```
% ./test
n (main) = 10
n (sub1) = 20
n (main) = 20
```

局所スコープ

- 関数内，ブロック内で参照できるスコープ．
- ある範囲に限定して，可視性の範囲を与える．
  - 入れ子にする（ネスト）ができることが多い．
  - ネストした場合，内側ブロックからは外側を参照可能なことが多い．
  - 兄弟は参照できない．

```
void a( void ) {
  AAA

  {
    BBB
  }

  {
    CCC
    {
      DDD
    }
  }
}
```

- AAA は，BBB,CCC,DDD から参照可
- BBB は他からは参照不可
- CCC は，DDD から参照可
- DDD は，他からは参照不可

ブロック

- 複数個の文などのコードのまとまり
  - 自然言語の段落に似ているが，入れ子構造を取ることができる.
- いろいろなスタイルがある:
  - {....} : C言語，Java 言語など
  - begin….end : Pascal, Ada など
  - オフサイドルール（字下げによる）: Python
  - 特定の構文による: BASIC，Fortran など.

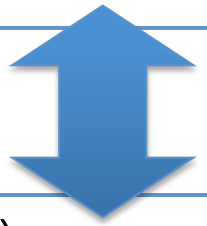
    例）Fortran の場合: IF. . . . END IF で囲む.
    - 任意のブロックを作成できないことに注意

いろいろなブロック

- ブロックのための特殊記号を使う言語では，コーディングスタイルによって，記述が一意に定まらない．

ブロック記号を使う言語（C）

```
if (i == 0){
  printf("AAA");
  if(j == 0){
    printf("BBB");
  }
}else{
  console.log("CCC");
}
```



インデントで区切る言語（Python）

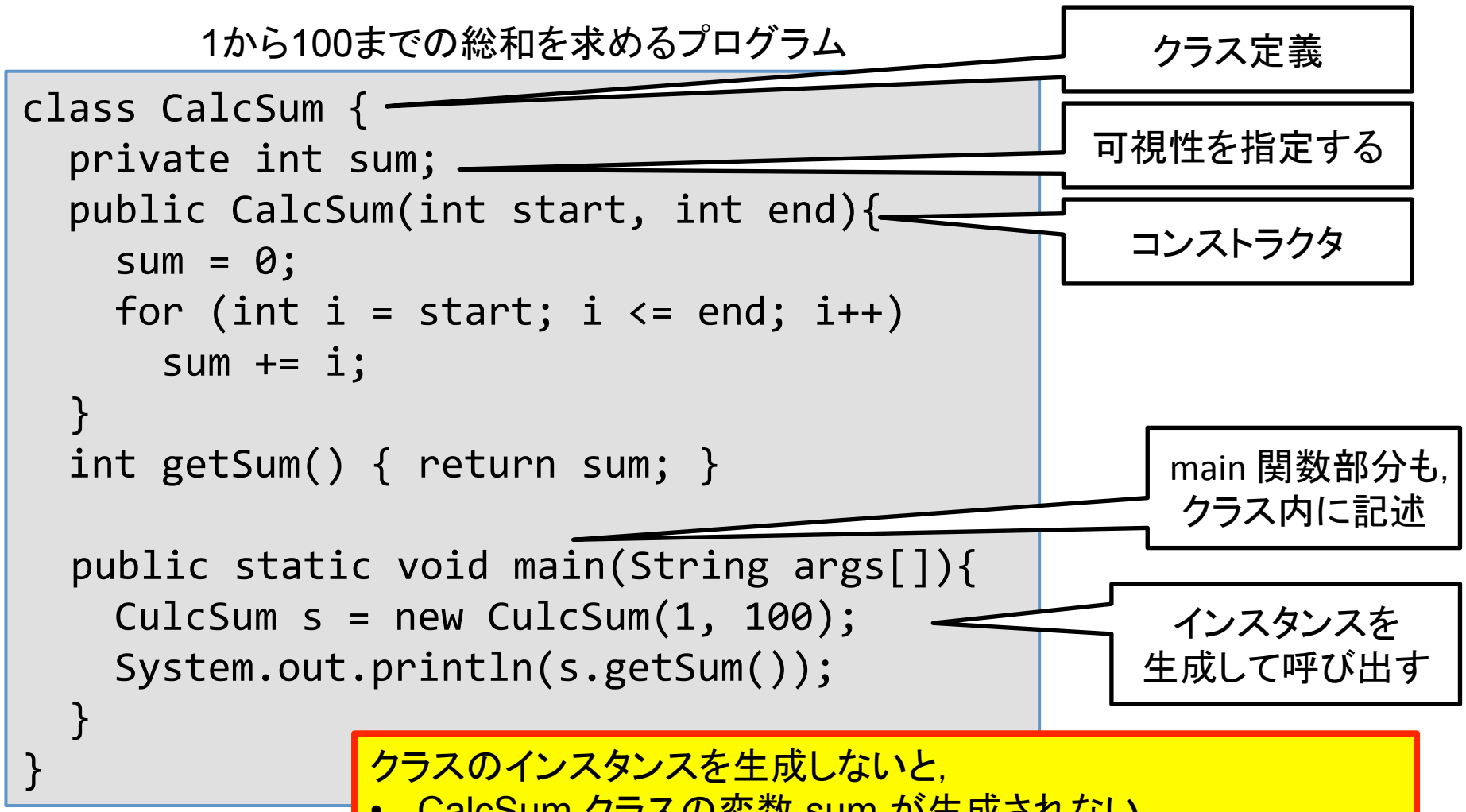
```
if i == 0:
  print "AAA"
  if j == 0:
    print "BBB"
else:
  print "CCC"
```

```
if (i == 0)
{
  printf("AAA");
  if(j == 0) {printf("BBB");}
}
else {
  console.log("CCC");
}
```

インススススコープ

- オブジェクト指向言語で，メンバ関数のみによって参照できるスコープ (カプセル化)
- クラスをインスタンス化（オブジェクトを作成，という意味)して，そのインスタンス（オブジェクト）の関数から参照できる.
- 別インスタンスでは，別の変数になる.
- 何を言っているかわからないので，例で説明.

Java言語 (以前見せた例)



クラスのインスタンスを生成しないと,

- CalcSum クラスの変数 sum が生成されない.
- 外部からは, インスタンス s 経由でないとアクセスができない.

クラススコープ

- オブジェクト指向言語で，クラス定義全体から参照できるスコープ
- クラスインスタンスに関係なく，一旦宣言すると，同一クラスでは同じ変数にアクセス.
- 別インスタンスでも，同一変数にアクセス.
- やっぱり何を言っているかわからないので，例で説明.

Java言語での例

クラススコープの例

```
public class Test {
  public static int a;
  public static void main(String[] args){
    Test t1 = new Test(); t1.add(1);
    Test t2 = new Test(); t2.add(4);
    mul2();
    System.out.println(a);
  }
  private void add(int i) {
    a += i;
  }
  private static void mul2(){
    a *= 2;
  }
}
```

static キーワードをつけると，
クラススコープになる

インスタンスメソッドには，
インスタンスを作成してアクセス

クラススコープでは，
インスタンスを作成せずに，
直接アクセス可能

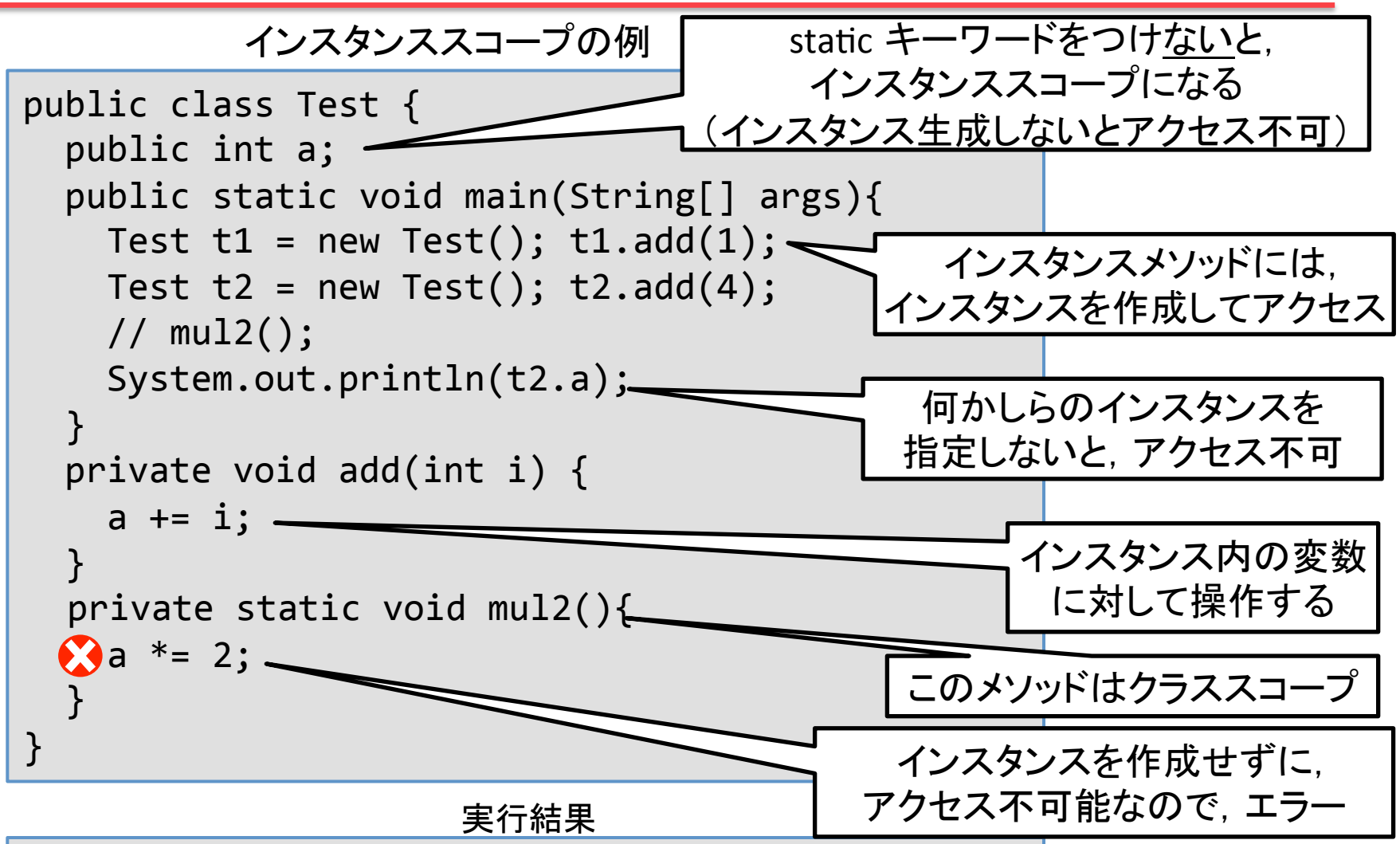
static キーワードをつけると，
メソッドもクラススコープになる

クラススコープでは，
インスタンスを作成せずに，
直接アクセス可能

実行結果

```
10
```

Java言語での例（比較のための例）

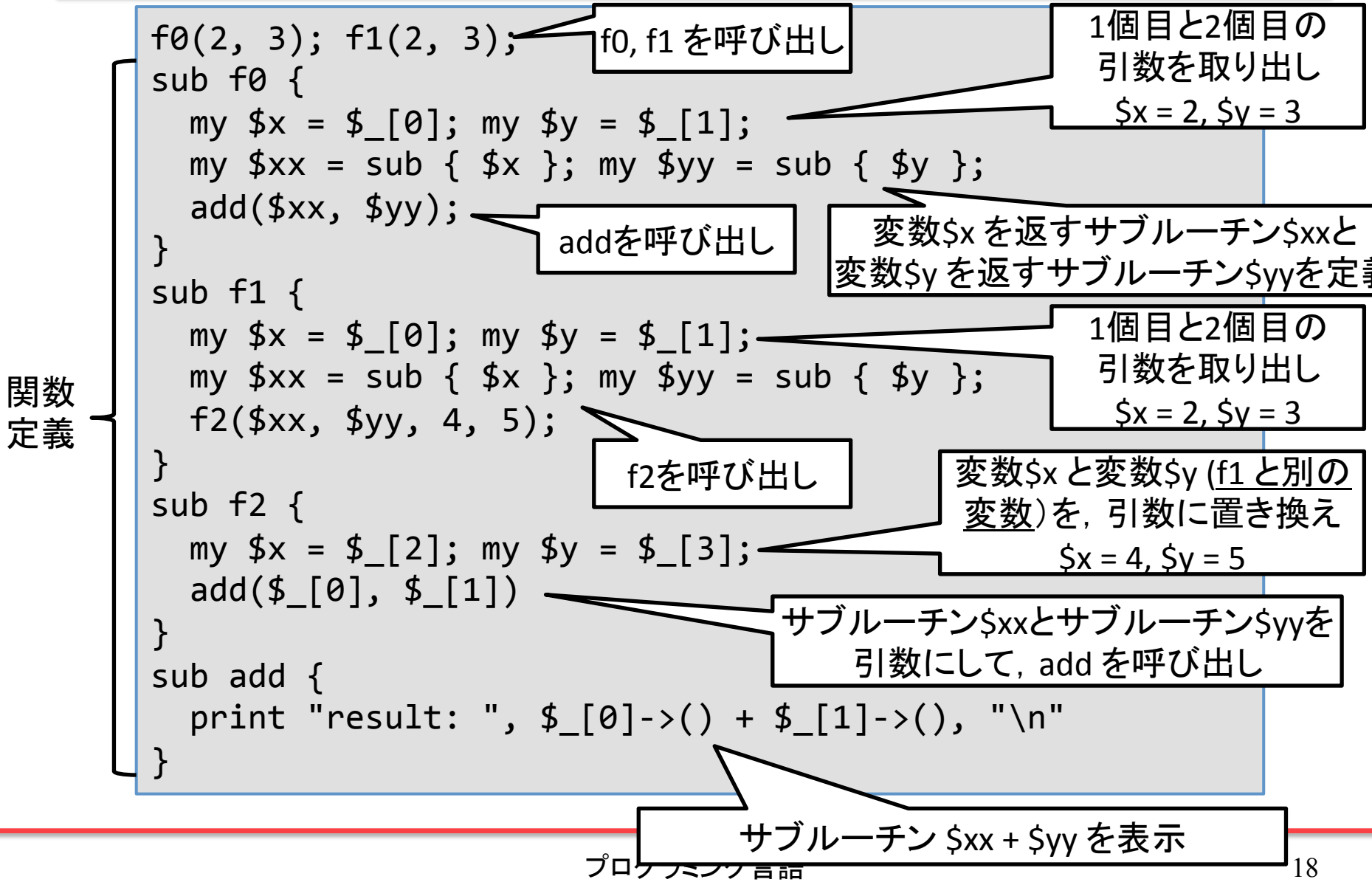


分類2：スコープ決定のタイミングによる分類

- 静的スコープ
  - 書かれた構文によって決定するスコープ
    - 構文のみによって決定
  - レキシカルスコープ（lexical: 字句的，辞書的な）
- 動的スコープ
  - 実行時に決定するスコープ
    - 実行時に，呼び出された側から呼び出し側のスコープを参照できる
  - ダイナミックスコープともいう
- ほとんどの言語が，静的スコープ
  - 動的スコープをサポートする有名な言語：
    - Emacs Lisp (emacs の設定ファイルを書いたり，拡張機能を書くための言語）
    - Perl (local キーワードを使用）
- 何が違うか？

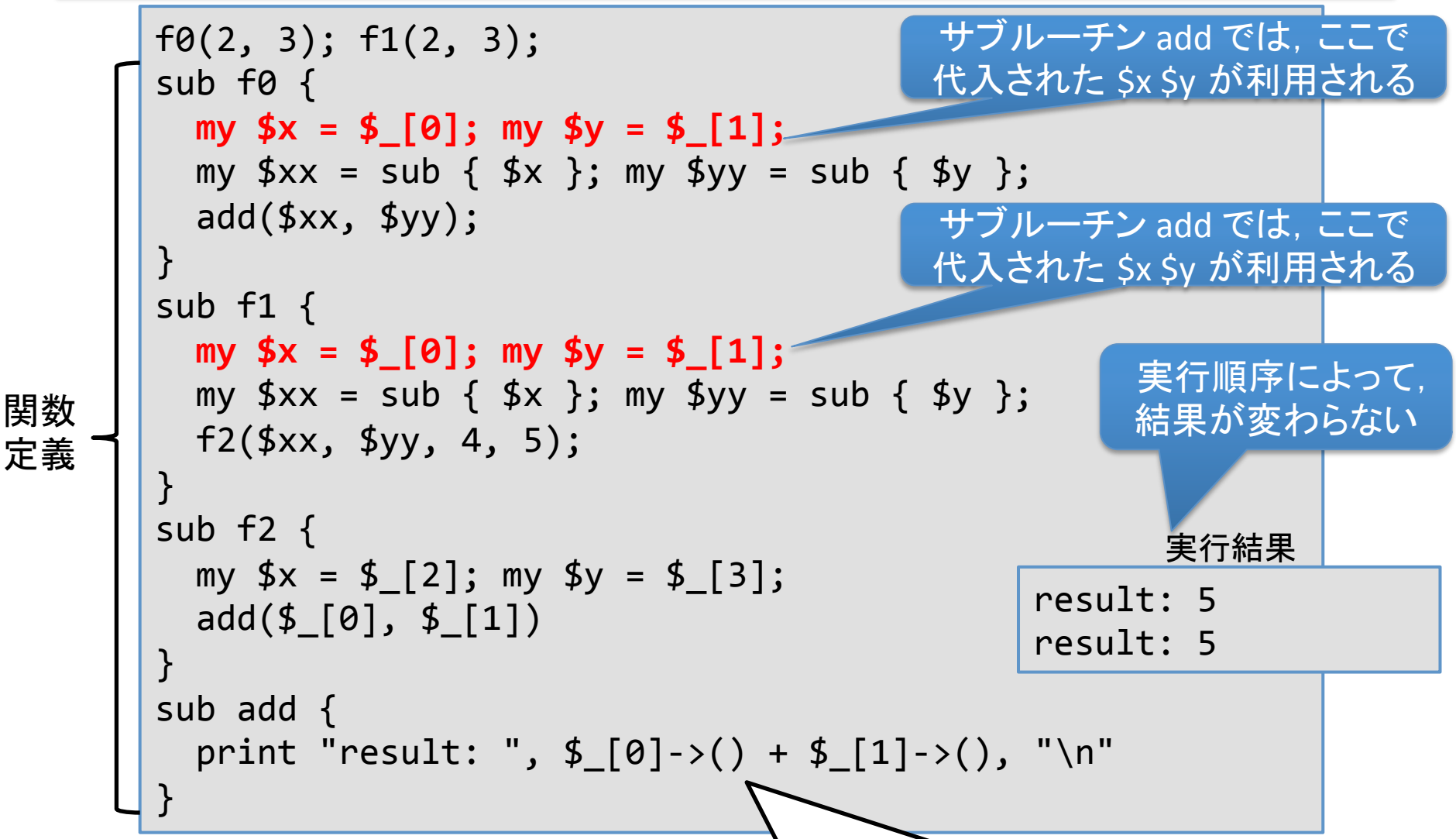
静的スコープの例 (perl)

実行順
f0 → add
f1 → f2 → add



静的スコープの例 (perl)

実行順
f0 → add
f1 → f2 → add

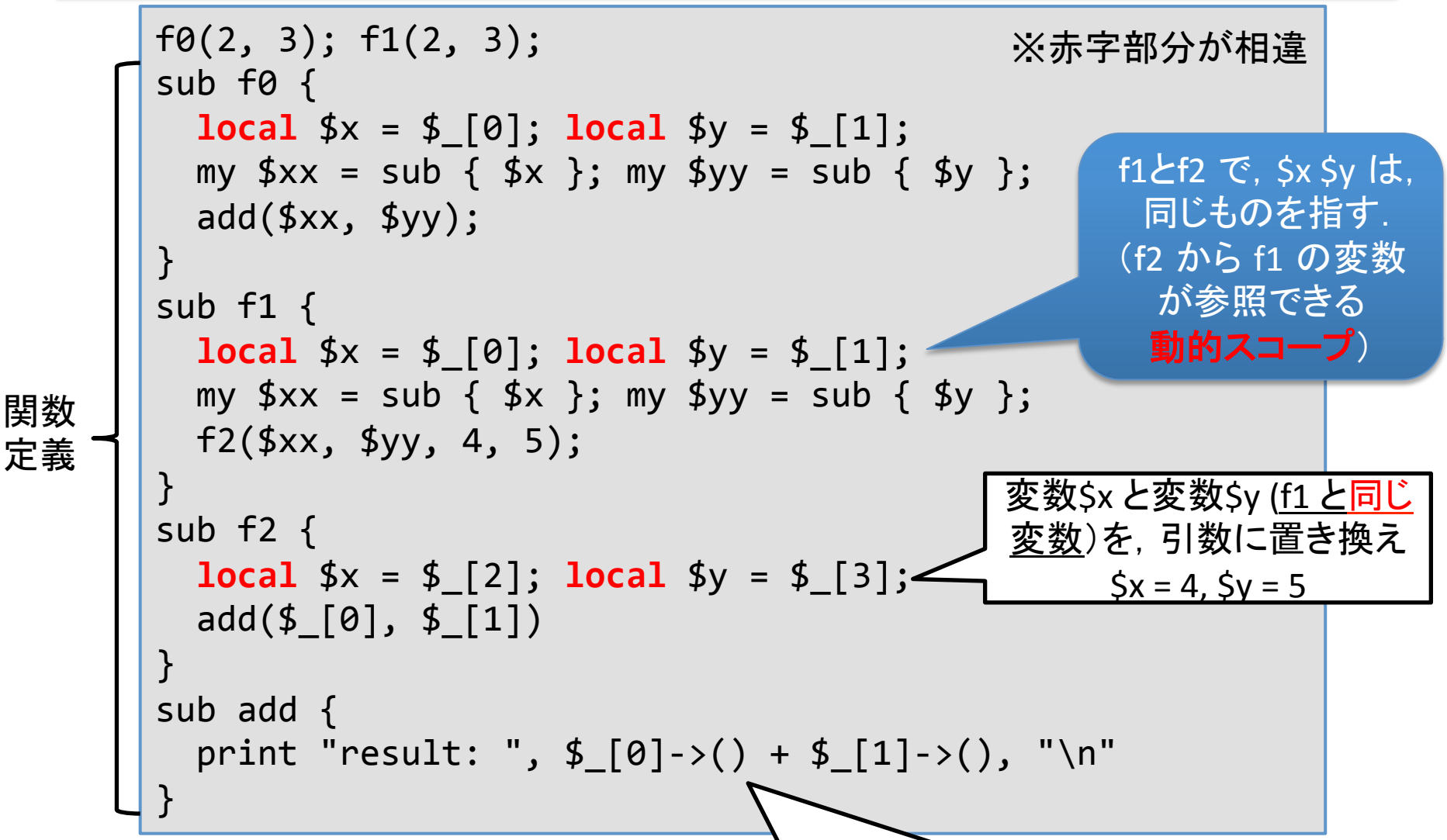


関数定義

```
f0(2, 3); f1(2, 3);
sub f0 {
  my $x = $_[0]; my $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  add($xx, $yy);
}
sub f1 {
  my $x = $_[0]; my $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  f2($xx, $yy, 4, 5);
}
sub f2 {
  my $x = $_[2]; my $y = $_[3];
  add($_[0], $_[1])
}
sub add {
  print "result: ", $_[0]->() + $_[1]->(), "\n"
}
```

動的スコープの例 (perl)

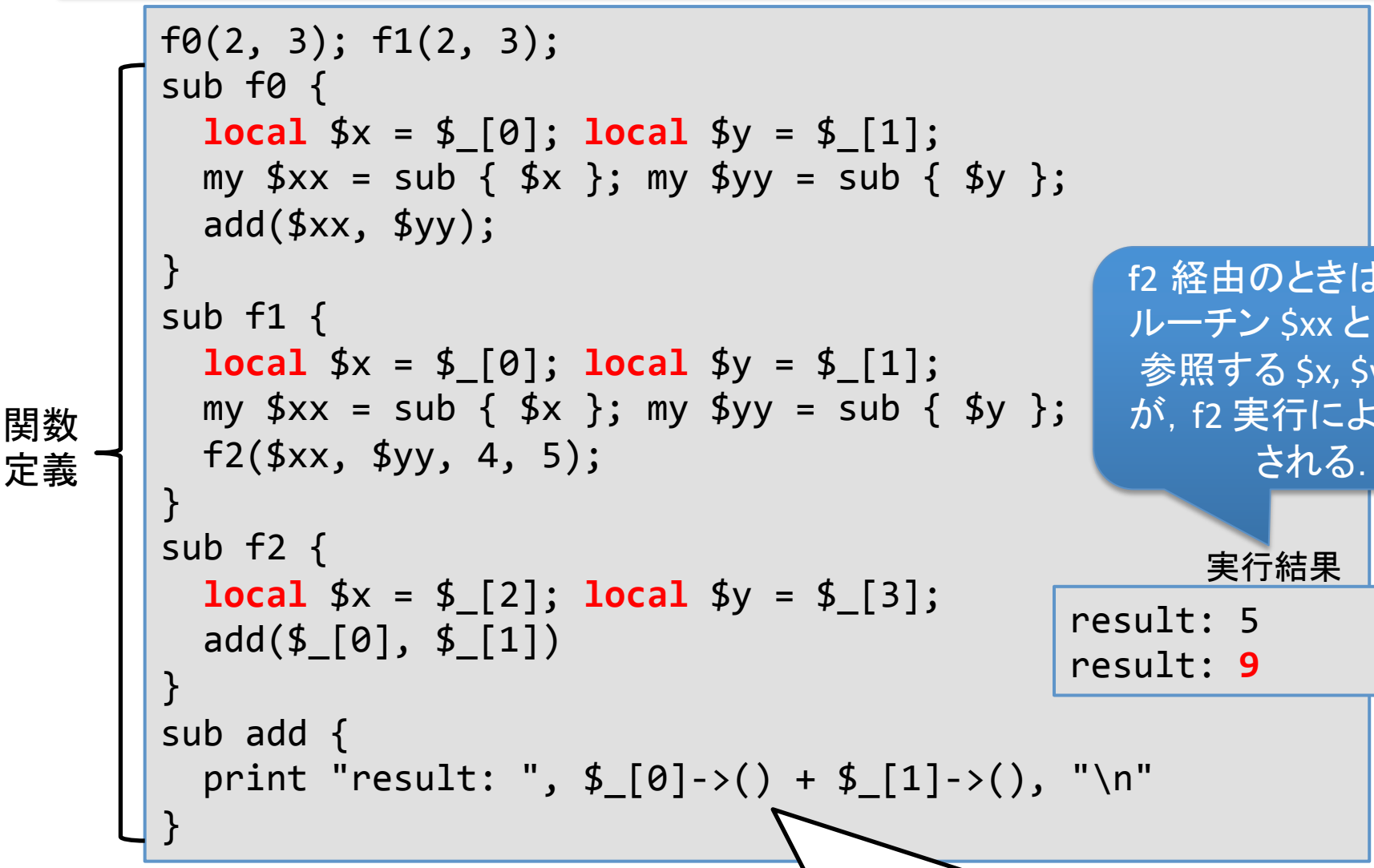
実行順
f0 → add
f1 → f2 → add



```
f0(2, 3); f1(2, 3);
sub f0 {
  local $x = $_[0]; local $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  add($xx, $yy);
}
sub f1 {
  local $x = $_[0]; local $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  f2($xx, $yy, 4, 5);
}
sub f2 {
  local $x = $_[2]; local $y = $_[3];
  add($_[0], $_[1])
}
sub add {
  print "result: ", $_[0]->() + $_[1]->(), "\n"
}
```

動的スコープの例 (perl)

実行順
f0 → add
f1 → f2 → add



関数定義

```
f0(2, 3); f1(2, 3);
sub f0 {
  local $x = $_[0]; local $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  add($xx, $yy);
}
sub f1 {
  local $x = $_[0]; local $y = $_[1];
  my $xx = sub { $x }; my $yy = sub { $y };
  f2($xx, $yy, 4, 5);
}
sub f2 {
  local $x = $_[2]; local $y = $_[3];
  add($_[0], $_[1])
}
sub add {
  print "result: ", $_[0]->() + $_[1]->(), "\n"
}
```

f2 経由のときは，サブルーチン $xx と $yy が参照する $x, $y の値が，f2 実行により更新される．

実行結果

```
result: 5
result: 9
```

サブルーチン $xx + $yy を表示

動的スコープはなんだったのか？

- 静的スコープ
  - 変数が，my で宣言（perl では，静的であることを示す）
  - 実行順により，結果が変わらない．
    - 同じ引数を利用して，$xx，$yy を呼び出す
- 動的スコープ
  - 変数が，local で宣言（perl では，動的であることを示す）
  - 実行順により，結果が変わる．
    - f2 の実行により，$xx，$yy が利用する変数 $x，$y が変更される．

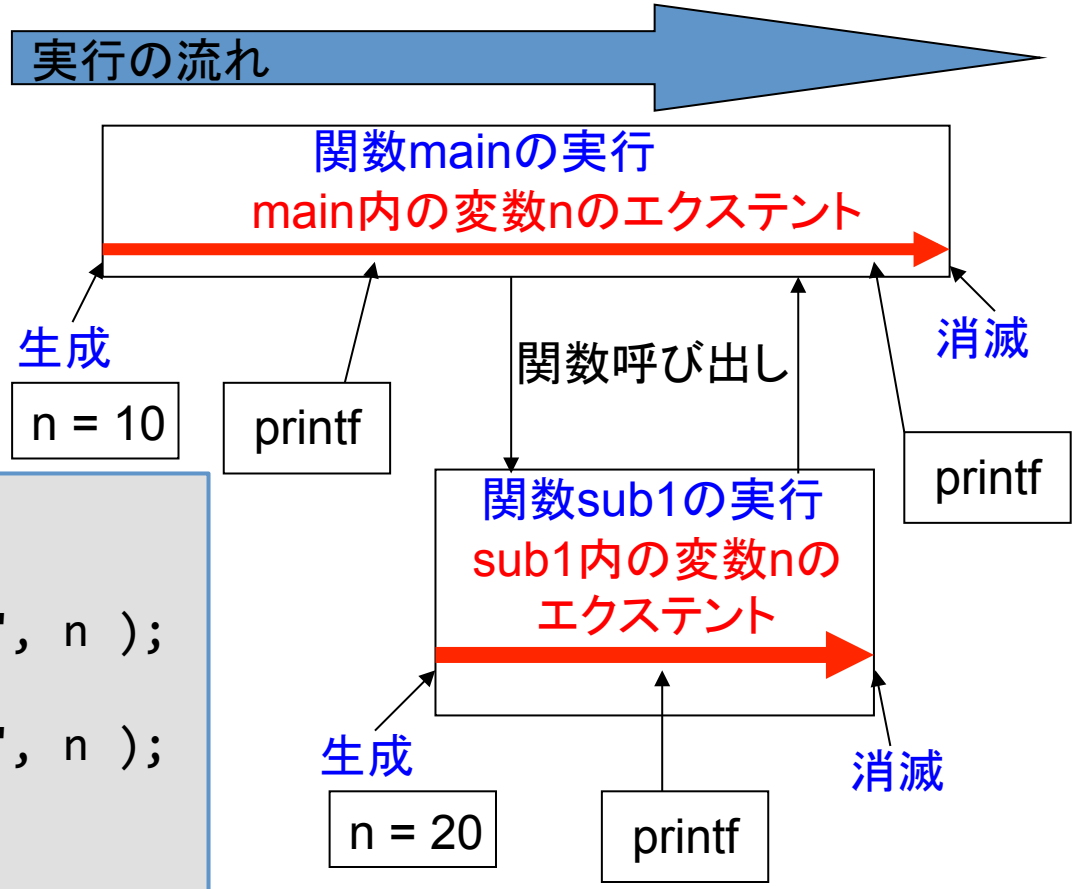
エクステント（extent）

- スコープは，アクセス可能な，場所的な範囲のこと

- それに対してエクステントは，アクセス可能な時間的範囲のこと
  - いつ生成され，いつ消去されるか？

自動変数の「エクステント」（変数の有効範囲）

- 局所変数 n が，どこで生成，消滅するか？




```
int main( void ) {
    int n = 10;
    printf( "n (main) = %d\n", n );
    sub1();
    printf( "n (main) = %d\n", n );
    return 0;
}
void sub1(void) {
    int n = 20;
    printf( "n (sub1) = %d\n", n );
}
```

スコープとエクステント

- スコープごとに有効範囲をまとめると，下のようになる．

種別	スコープ範囲	エクステント
局所（自動）スコープ	ブロック内	ブロックに入った時点から出るまで
大域スコープ	プログラム全体	プログラム開始から終了まで
ファイルスコープ	ファイル内	プログラム開始から終了まで（コンパイルして一つのプログラムになるため，開始時に生成される）
インスタンススコープ	アクセス修飾子で指定した範囲（private/protected/public 等）	オブジェクトインスタンス生成から消滅まで（Java では，ガベージコレクタによる回収まで）
クラススコープ	アクセス修飾子で指定した範囲（private/protected/public 等）	クラスロードから，プログラム終了まで（動的にクラスがロードされるため，利用開始時点で生成される）

手続き

- 実行すべき一連の計算ステップのまとまり
  - 言語により，様々な呼び名がある．
    プロシージャ，ルーチン，サブルーチン，メソッド，
    関数など．
- プログラム実行中の任意の時点で呼び出すことができる．
  - 他の手続きからの呼び出しも，自分自身からの呼び出し（再帰呼び出し）もありうる．

手続きとモジュール

- モジュール：名前をつけて，入出力を明確化した手続きのこと.
  - まとまりを持ったソフトウェア部品
  - インタフェース（入力と出力）と，実装を分離することで，ソフトウェアの設計を容易にする
- モジュール性（Modularity)：あるモジュールが他のモジュールに依存する度合い
  - インタフェースが定義され，それ以外が関連しなければ，モジュール性が高い
    ＝部品として再利用可能

スコープとモジュール性

- C言語のプログラム
  - 大域スコープの変数を多用した場合：モジュール間で変数を共用することが多くなる→ モジュール性：低.

```
int a, b;
int main(void) {
    a = 2, b = 3;
    printf( "[交換前] a = %d, b = %d\n", a, b );
    swap();
    printf( "[交換後] a = %d, b = %d\n", a, b );
    return 0;
}
void swap() {
    int copy;
    copy = a;
    a = b;
    b = copy;
}
```

呼び出し元と呼び出し先の依存性が高い
＝モジュール性：低

swap を別プログラムで利用したい場合：
- 大域変数 a, b も同時に別プログラムに移す必要がある
- 呼び出し元のプログラムで，変数 a, b を利用する必要がある

モジュール性の高いプログラム

- C言語のプログラム
  - 大域スコープの変数の利用を避け，外部との値のやりとりを引数，返戻値に限る→モジュール性：高．

```
int main(void) {
    int a = 2, b = 3;
    printf( "[交換前] a = %d, b = %d\n", a, b );
    swap( &a, &b );
    printf( "[交換後] a = %d, b = %d\n", a, b );
    return 0;
}
void swap( int *a, int *b ) {
    int copy;
    copy = *a;
    *a = *b;
    *b = copy;
}
```

呼び出し元と呼び出し先の依存性が低い＝モジュール性：高

swap を別プログラムで利用したい場合：
- 関数部分だけコピーすれば，利用可能．変数が他に依存しない．
- 呼び出し元のプログラムで利用する変数にも制約がない

パラメータへの値の代入

- Call-by-value（値渡し）
  - 関数を呼び出す時に，パラメータに入れられた値を評価してから，呼び出す方式．
  - foo(bar(a)) という呼び出しの時，bar(a) を呼び出し実行してから (仮にxと置く) その結果を元に，foo(x) を呼び出す．
- Call-by-reference（参照渡し）
  - 関数を呼び出す時に，パラメータに値への参照を入れて呼び出す方式．呼び出し先は，参照を使って値を直接入出力する．
  - C 言語のポインタ（前のスライドの swap 関数）．
- Call-by-name（名前渡し）
  - 関数を呼び出す時に，パラメータに記述された変数名などの名前をそのまま渡し，パラメータが関数内で初めて利用される時に評価する方式（C言語では存在しない）．

Call-by-value と Call-by-name の比較

- Call-by-value
  - 関数を呼び出す時に，パラメータに入れられた値を評価してから，呼び出す方式.
- Call-by-name
  - パラメータが関数内で初めて利用される時に，評価する方式
  - C言語やJavaでは存在しない.
  - 遅延評価の戦略で評価される
    - 遅延評価：実際の計算を，値が必要になるまで行わないこと. ⇔ 先行評価（Call-by-value の戦略）
    - 必要になるまで計算しないため，引数が利用されない時は，計算が行われない.
  - 計算量の最適化を目的とした言語に存在.
    - 例：Scala 言語 (Java に似た，オブジェクト指向言語) .

Call-by-value と Call-by-name の比較例

※main は省略

Call-by-value の例 (Java)

```
public String sayHello(){
 System.out.println("in sayHello()");
 return "HELLO";
}
public void print(String s){
 System.out.println("in print()");
 System.out.println(s);
}
public void printHello(){
 print(sayHello());
}
```

printHello を呼んだ場合の実行結果

```
in sayHello()
in print()
HELLO
```

Call-by-name の例 (Scala)

```
def sayHello() = {
  println("in sayHello()")
  "HELLO"
}
def print(s : => String) = {
  println("in print()")
  println(s)
}
def printHello() = {
  print(sayHello())
}
```

printHello を呼んだ場合の実行結果

```
in print()
in sayHello()
HELLO
```

- 評価順序が異なることに注目！

返戻値

- 関数の返り値は，通常は一つしかない
  - （明確な理由はないが，おそらく）数学の関数 y=f(x) の概念に起因する
- 複数の値を返却したい時はどうするか？