プログラミングIII

黒瀬　浩

kurose@neptune.kanazawa-it.ac.jp

OH: 講義の前後，火4限21・405

居室　67・121

レポート3 （4点） ソースと結果をA4 1枚(両面可) 10回開始時提出

親からスレッドを30個作る（スレッドを管理する配列を用意する）
各スレッドを開始させ，全ての終了を待つ
終了メッセージProgram endを出力する
各スレッドでは
スレッドのIDを取得，開始時刻を取得，乱数で2から10秒待ち，
終了時刻を取得し，スレッドID，開始時刻，待ち秒数，終了時刻を表示
参考
スレッドID取得　Thread.currentThread().getId()
現在時間取得 java.time.LocalTimeのLocalTime.now()
時刻書式指定 java.time.format.DateTimeFormatterのofPattern()
整数値nまでの乱数 randomインスタンスのnextInt(n)
整形して表示 System.out.prinf(書式指定，表示項目)
出力例

```
    23 start 15:48:11    wait    2 end 15:48:13
    13 start 15:48:11    wait    3 end 15:48:14
    (中略)
Program end
```

注：各スレッドは平行動作させること（全てのスレッドを作成し終えてから開始させる）

ガーベッジコレクション（ゴミ集め）

C言語で動的にメモリ領域を取る

```
*ptr = malloc(メモリサイズ);      // メモリをサイズ分確保
 ptrの先に値の設定参照を行う
 free(ptr);                       // 確保した領域の解放
```

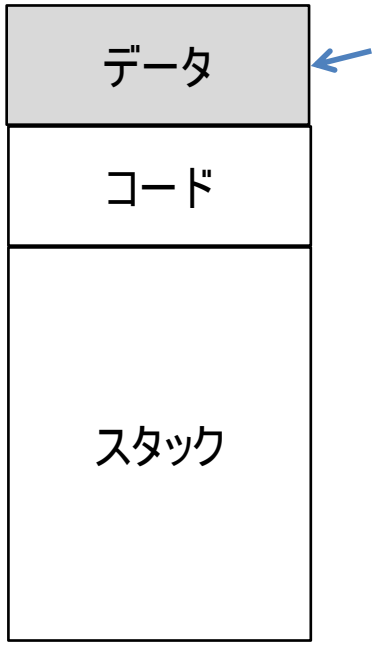
上記は動的に増減するデータ（例えば構想的なデータ）に対して使う
C言語では構造体で必要なデータ構造を定義し，必要な分malloc/allocする

freeし忘れるとメモリ領域が無駄になる
いつかは領域が足らなくなる
すでにfreeした領域をアクセスすると異常な動作となり強制終了する

空き領域はあるが必要な分だけ連続した領域が取れないことがある
(実行継続不可能)
変数，メソッドも静的なもの(クラス）と動的なもの（インスタンス）がある

参考：スタティック変数

動的な変数はスタックに置かれるのでreturn後はアクセスできない
常にアクセスできる変数、定数はスタックでなく別領域に置かれる



グローバル(大域)変数
静的変数

グローバル(大域)変数
どこからも参照可能な変数
プログラム開始位置からの相対番地は固定
言語により参照のみ、更新可能の属性を持つ

静的(static)変数
プログラム開始から終了まで存在
プログラム開始位置からの相対番地は固定
他の関数に見せない場合もある

```
public static int a=10;
public static void main(){ ... }
```

グローバル、静的

C言語のグローバル変数、関数はどこからも参照可

参考：スタックの使用例

一般的言語で関数(メソッド）呼び出しを考える

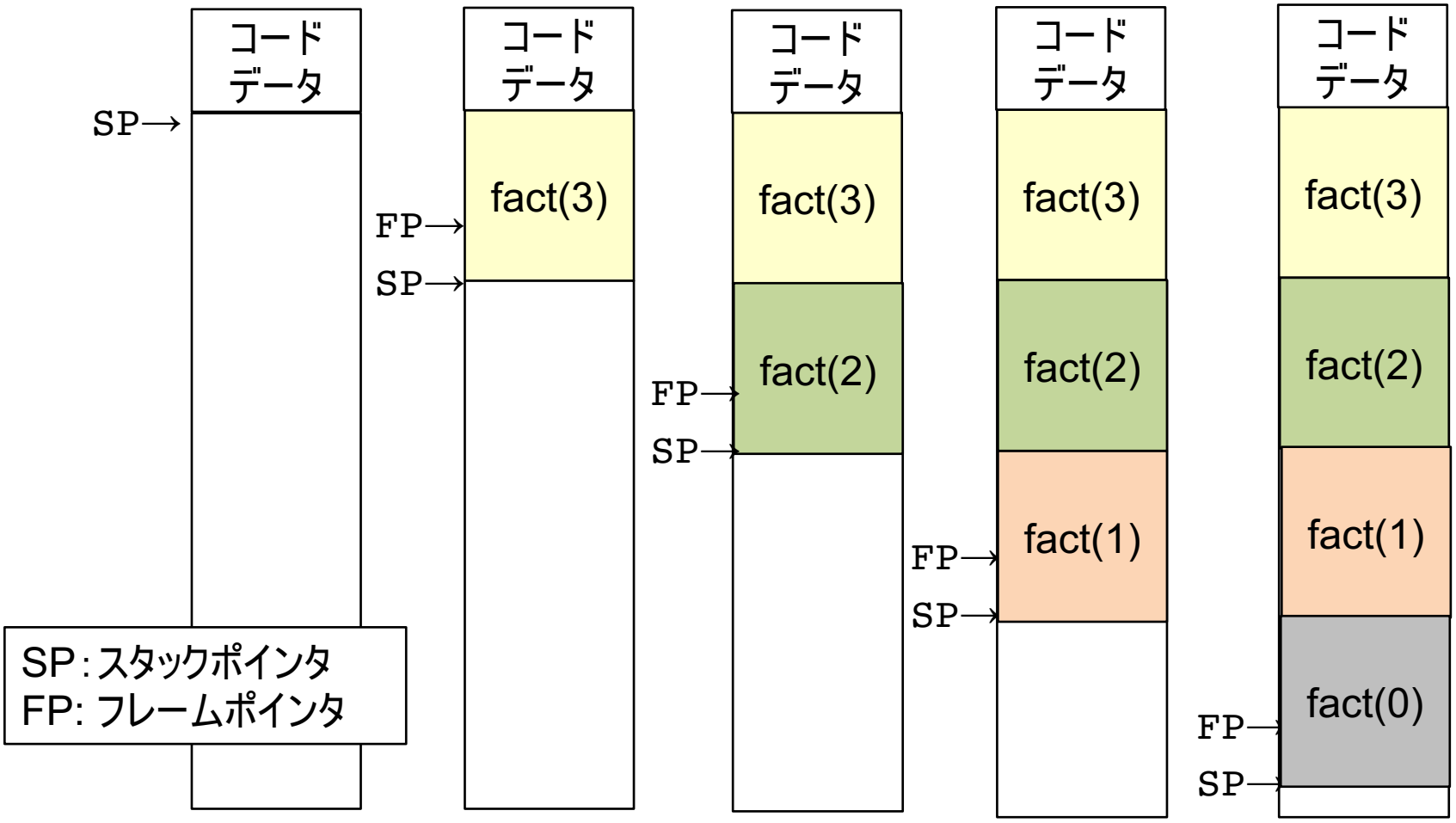
関数factは引数が0なら1を返し
そうでなければn-1でfactを呼ぶ（再起呼び出し）
pythonでの実装例

```
def fact(n):
    return n*fact(n-1) if n!=0 else 1
print( tact(3) ) # 3!
```

変数nがプログラム空間上のある場所1つだとこのプログラムは実現できない

関数が呼び出されるごとに変数(この場合はn)が生成されなければならない
return したらその変数は以降使われない

参考：スタックの動的変化　階乗の再帰呼び出しの例



プログラムのアドレス空間はOS、ハードウェアに依存する
アドレス空間32bitで仮想記憶が使えるOSでは4GB
スタックがアドレス空間を超えたらスタックオーバーフローで強制終了
(後述するヒープ領域もメモリ空間を使う)

参考：ヒープ(heap)領域

関数内のデータはreturnすると使えなくなる
関数の呼び出し、復帰に関わらず生存するオブジェクト
　　ファイル記述子(pythonの例)

```
f=open('f1.txt', 'w')
f.write(xxx)
f.close()                    # ここでfは消滅
```

ファイルオブジェクト、クラスオブジェクトは、関数の寿命とは別
C言語での管理

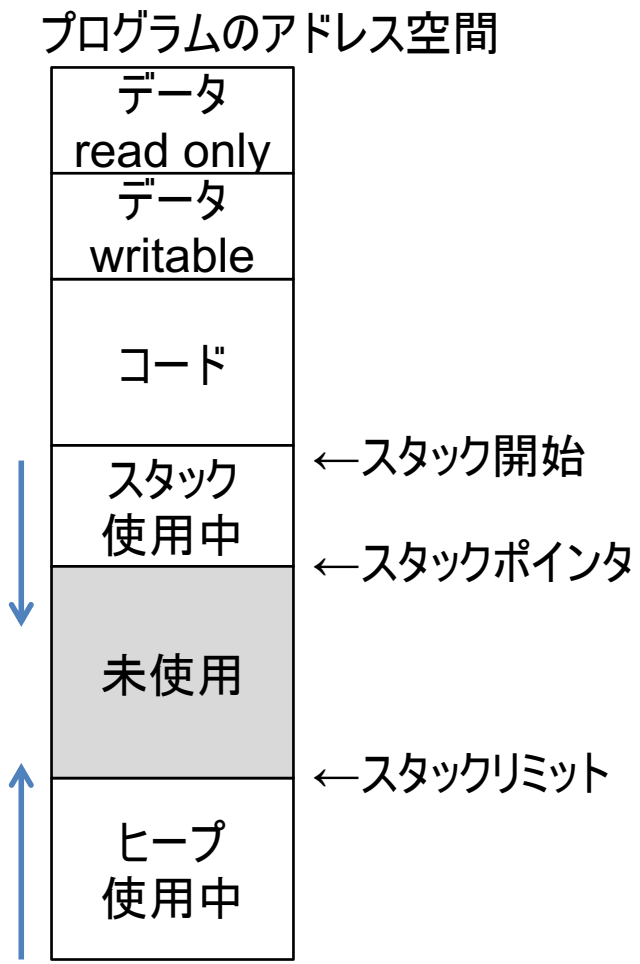
```
変数=malloc(サイズ)                    // メモリ領域を確保する
変数への代入・参照を行う
free(変数)                             // メモリから解放する
```

オブジェクト指向

```
インスタンス変数=new クラス名()        // 生成
クラスオブジェクトでreturnするか、他からdestroy,killで消滅
```

参考：メモリ配置

スタックはpush()で成長、pop()で縮小なので、伸び縮みするだけ
ヒープはアプリケーションの都合で使用、解放が行われる
=> スタックとは別の領域が必要

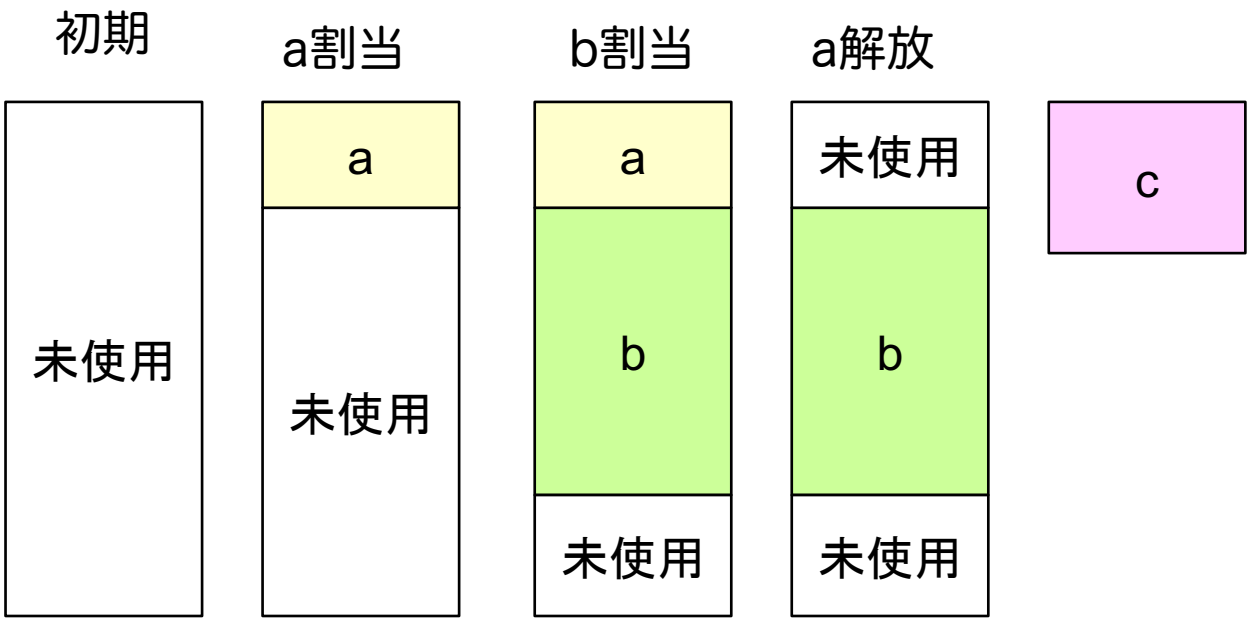


スタック、ヒープはどれだけ使うかわからない
多くの機種では、
　　スタックはアドレスの小さい方から
　　ヒープはアドレスの大きい方から
割りあてる
重なってしまったら、これ以上実行できない
　　stack overflow, heap overflow
のエラーを出して強制終了させられる

左図でスタックポインタ，スタックリミットは
状況により動的に変動する

参考：ヒープ領域の使用方法

ヒープに確保するサイズや寿命はまちまち
合計サイズが余っていても確保できないことも発生する



cを割り当てたいが
連続領域がとれない

分割すれば確保可能

分割するとプログラムに不都合が発生

変数は先頭からの相対位置（オフセット）でコード生成されているため
分割されるとプログラムが全然関係ない別のオブジェクトの部分を
アクセスしてしまう

バディ（Buddy）アルゴリズム

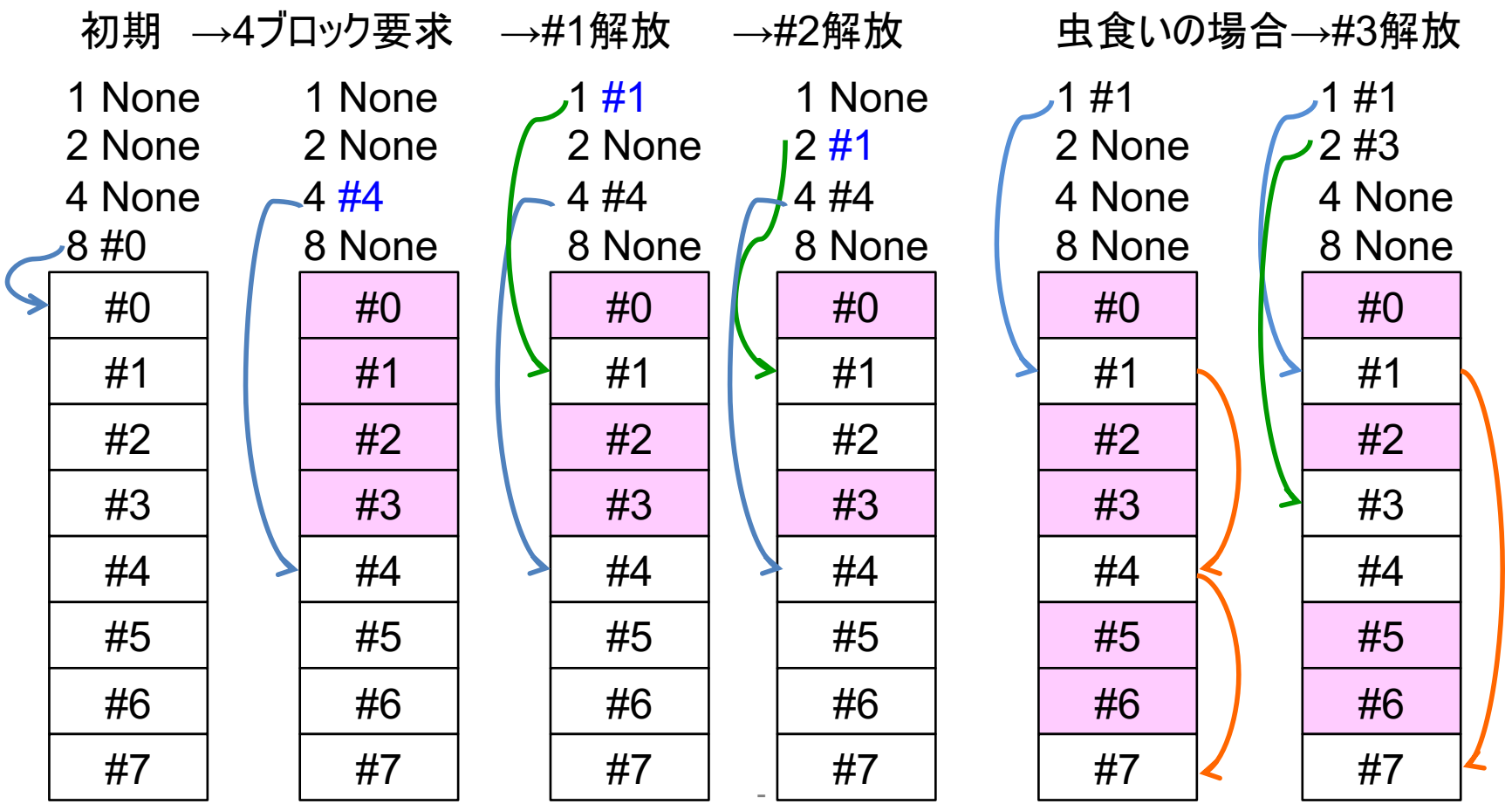
1963 by Harry Markowitz, and was first described by Kenneth C. Knowlton (published 1965).

メモリ領域を有効に確保する方法の一つ

領域の分割最小単位(ブロックという)を設定する

必要とするサイズが入る最小の2^nブロックを確保する

サイズごとにエントリをリンク。解放時、前後が未使用だったら結合



ガーベッジコレクションの必要性

C言語ではユーザが確保した領域はfree()してメモリを解放する

Javaではインスタンス終了時に不要な部分を解放するが
その都度，解放していると動作が遅くなる
そこで，
メモリが不足してきたら，Java VMが未使用部分をまとめて解放する

いつどのように解放するかはVMやOSにおまかせだが
途中でガーベッジコレクションを起こしたく無いこともある
（リアルタイム処理で遅延を許容できない場合）

以下で強制的にガーベッジコレクションを起こさせることができる

```
Runtime.getRunrime().gc();
```

ArrayList

配列は追加や削除が動的に行えない

作成　ArrayList<型> 変数名 = new ArrayList<型>();

メソッド

追加　add(型　値)

取得　get(int 要素番号)

削除　remove(int 要素番号)

要素数 size()

リスト5-1抜粋

```
ArrayList<String> months = new ArrayList<String>(); // []
months.add("Jan");        // ["Jan"]
months.add("Feb");        // ["Jan", "Feb"]
months.add("Mar");        // ["Jan", "Feb", "Mar"]
months.remove(1);         // ["Jan", "Mar"]
months.size()      // 2
```

ラッパークラス　基本型を包むクラス

基本型	ラッパークラス
boolean	Boolean
byte	Byte
char	Charcater
short	Short
int	Integer
long	Long
float	Float
double	Double

既に出てきた　Integer.parseInt()　はintラッパークラスのメソッド

List5-2抜粋

```
ArrayList<Integer> integerList = new ArrayList<Integer>();
integerList.add(new Integer(50)); // Integerクラスに50を渡す
Integer interger0 = integerList.get(0);    // 型はInteger
int i0 = integer0.intValue();              // 整数化
```

コレクションフレームワーク

リスト

- 要素の順番を管理する
- 要素の取り出し，追加，削除
- `ArrayList, LikedList`

マップ

- キーと値(Key Value)
- 他言語のハッシュ，辞書と同じ
- `HashMap, LinkedHashMap`

セット

- 要素に重複が起きない
- 他言語のセット，集合と同じ
- `HashSet, TreeSet`

`LinkedList`は，要素から次の要素の位置と前の位置を管理している
前からたどる場合，後ろからたどる場合，要素の追加，要素の削除で高速

HashMapの例

```
HashMap<String, String> map = new HashMap<String, String>();
map.put("key1", "val1");     // 要素の追加
map.put("key2", "val2");
map.entrySet();          // キーと値のペアの一覧取得
// printすると  [key1=val1, key2=val2] 順番は保証されない

map.values();            // 値一覧取得
// printすると  [val1, val2]  順番は保証されない

map.keySet();            // キー一覧取得
// printすると  [key1, key2]  順番は保証されない

map.get("key1");         // キーがkey1の値を取得, キーがなければnull
// printすると  val1
```

値を指定してキー一覧を得たい場合は？
既に存在するキーに対してput()すると？

イテレータ(Iterator) 反復子

一つづつとり出す

```
HashSet<String> set = new HashSet<String>();
set.add("A");
set.add("B");
Iretator<String> it = set.iterator();
while(it.hasNext()){
	String str = it.next();
	System.out.println(str);
}
```

キーボードから読む

```
Scanner sc = new Scanner(System.in);
sc.next();  // イテレータと同じ
```

拡張for文 ひとつづつ取り出して処理する

```
for(型　変数 : コレクション){　　}
```

pythonのfor文

```
for 変数　in コレクション:
    ブロック
```

phpのforeach文

```
foreach ( 変数　as 配列 ){　　}
```

List5-7抜粋改

```
String [] months={"Jan", "Feb", "Mar"}
for(String str : month){
    System.out.println(str);
}
```

キュー，スタック

キュー
　　末尾に追加，最初から取り出し(先入先出し,FIFO)

スタック
　　末尾に追加，末尾から取り出し(後入先出し,LIFO)

リストを使えばキュー，スタックを実装できる

キュー：　　待ち行列，構造探索
スタック：　演算の計算，関数呼び出しの管理，構造探索，章番号付け

グラフ（点と線からなる構造）
　　探索するときに再帰呼び出しやキュー，スタックを使う

高階関数(higher-order function)

関数に関数を渡したい場合がある

```
# pythonの例
import math as M   # 数学パッケージインポート
def f(g, p):       # 関数定義
    return g(p)    # 引数gを関数として実行

print(f(M.sin, M.pi))    # 関数fにsin()とM.piを渡す
print(f(M.cos, M.pi))    # 関数fにcos()とM.piを渡す
```

結果

```
1.2246467991473532e-16   # sin π
-1.0                     # cos π
```

オブジェクト指向では，関数，クラス，変数，メソッド，定数もオブジェクトとして扱いたい（全てがオブジェクト）

内部クラス リスト6-1

クラスの中にクラスを定義する

```
class Outer {
    private String message = "*Outer";
    void doSomething(){
        class inner {
            void print(){
                System.out.println("*Inner");
                System.out.println(message); // class外
            }
        }   // doSomething()の中でインスタンスを作成している
        Inner inner = new Inner();
        inner.print();
    }
}
public class InnerClassExample {
    public static void main(String[] args){
        Outer outer = new Outer();
        outer.doSomething();
}
```

匿名クラス　名前のないクラス List6-1から

```
interface SayHello {
   public void hello(); // 抽象メソッド
}
class Greeting {
       // SayHelloインターフェースを実装するクラスを引数として受け取る
   static void greet(SayHello s){  // (2) 渡されたpをsに
       s.hello();           // (3) p.hello()と同じ
   }
}

class Person implements SayHello {
    public void hello(){       // (4) 最終的にここが呼ばれる
        System.out.println("Hello");
    }
}
public class SimpleExampe {
    public static void main(String[] args){
        Person p = new Person();
        Greeting.greet(p);   // (1) Person型のpインスタンスを渡す
    }
}
// クラスを限定せずにそのクラスに応じたメソッドを呼び出す
```

演習

5章の例題

5章の章末問題

- または　レポート3