データ構造とアルゴリズム
（第1回）

モバイルコンピューティング研究室
柴田史久

1

1

教科書 第1章（pp.3～9）

アルゴリズムとは？

2

2

はじめに

- アルゴリズムは処理の手順を記述したもの
- プログラム：
  - アルゴリズムをプログラミング言語で表現
- ここでは
  - Java言語をマスターしていることを想定
    - Java言語の約束事（オブジェクト指向）を理解
    - 行いたい処理をJavaプログラムとして実装可能

3

3

プログラミングをマスターするには

1. 問題を分析し，どんなプログラムを書くのかを決める（プログラムの仕様の決定）
2. 使用するアルゴリズムとデータ構造を選択
3. 実際にプログラムを書く

3.については「プログラミング言語」「プログラミング演習２」で修得済み

↓

本講義と「プログラミング演習２」で2.についての修得を目指す

4

4

データとは

- 現象や性質を何らかの枠組みに従って形式化したもの
- 具体例：数値，文字列，画像，etc…

24.5, 26.5, 25.7, 24.9, …

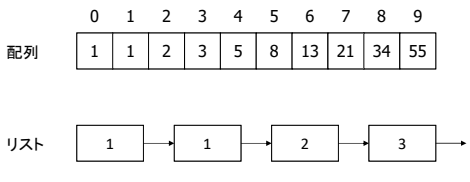
氏名	電話番号	性別	年齢
立命太郎	077-566-1111	男	105
衣笠花子	077-561-2600	女	24
:	:	:	:

5

5

データ構造とは

- コンピュータのメモリ上（ディスク上）のデータの並べ方
- １つ／複数のデータを編成し保持する構造
- 例：配列，リスト，木，etc…

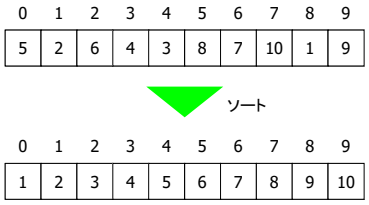


6

6

アルゴリズムとは

- データ構造中のデータを操作する手法・手順
- １つ／複数のデータを操作し目的の結果を得るための一連の処理手順
- 例：整列（ソート），探索，etc…



7

7

アルゴリズムとデータ構造の関係

- アルゴリズム＋データ構造＝プログラム
  - Niklaus Wirth（ヴィルト，Pascalの設計者）
- 「アルゴリズム＝プログラム」ではない
- 当初のアルゴリズムは数学上の問題の解法
- コンピュータの登場でプログラムとしての性質を帯びるようになってきた

8

8

チューリングマシン

- コンピュータの数学的なモデル
- 記憶装置として無限長のテープを有する
  - 記憶容量は無限大
- 動作速度については規定なし（※注）
  - 動作速度は無限大とみなされていた

※注：チューリングマシンが考案された時点ではコンピュータは存在せず
アルゴリズムの実行時間という概念が希薄

9

9

現実のコンピュータ

- 計算能力，記憶容量ともに有限
- アルゴリズムの性能を考える必要あり
  → 第2章　計算量

10

10

時間と空間のトレードオフ

- どちらのアルゴリズムが良い？
  - 遅いがメモリをあまり使わないアルゴリズム
  - 速いがメモリをたくさん使うアルゴリズム
- 例：
  - 補助的な情報を事前計算すれば速度は向上
  - 補助的情報を保存するメモリが必要

データをどのように表現するかが非常に重要

↓

アルゴリズム＋データ構造＝プログラム

11

11

なぜアルゴリズムを勉強するのか?

- 好奇心
- プログラミング技術の向上
  - 部品化されているなら使い方を知れば十分？
- 同じ処理をする複数のアルゴリズムが存在
  - 何らかの基準で選択する必要
    - 入力データや使用できる資源（CPU，メモリ）を考慮
  - アルゴリズムの性能・ふるまいを「知る」必要

12

12

教科書 第2章（pp.10～30）

計算量

13

13

アルゴリズムの性能の基準

- 直観的には実行時間で計測
  - 複数のアルゴリズムを集めて実行時間を比較
  - アルゴリズムのベンチマークテスト
- 一定の条件下で実施する必要あり
  - どんなマシンを使用するか
    - CPUのアーキテクチャで得意・不得意な処理が存在
  - コードを書く人の腕前で左右される
  - プログラミング言語によっても異なる
  - コンパイラの性能によっても異なる

実験的なアプローチでは評価できない

14

14

計算量(1)

- 性能評価には実在のマシンは使わず
  計算量（complexity）という尺度を利用
  - 仮想的なコンピュータによるアルゴリズム実行時間を入力の大きさ n の関数として表現したもの
- 例：実行時間が入力の大きさ n の2乗に比例
  - 実行時間が $O(n^2)$ のアルゴリズム
  - オーダー n^2 と読む

15

15

オーダー記法

- ある関数 $f(n)$ に対して，計算量 $T(n)$ が $O(f(n))$ であるとは，ある正定数 c と n_0 が存在し，n_0 以上のすべての n に対して

$$T(n) \leqq c f(n)$$

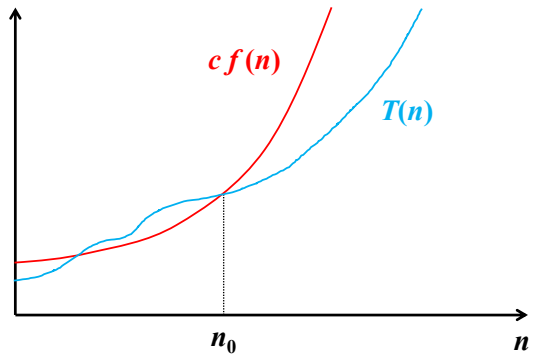
が成り立つことをいう

- n が十分大きいところで計算量を漸近的に評価
- 主要項の係数を除いたもの
- 評価は入力サイズ n が十分大きなときに成立

16

16

オーダー記法



17

17

オーダーの算出例(1)

- 計算量 $T(n)$ が，$T(n) = 4n^3 + 2n$ とする．
  $n \geqq 1$ なるすべての n に対して

$$\begin{aligned} T(n) &= 4n^3 + 2n \\ &\leqq 4n^3 + 2n^3 \\ &= 6n^3 \end{aligned}$$

が成り立つので $T(n)$ が $O(n^3)$ とわかる

18

18

オーダーの算出例(2)

- 計算量 $T(n)$ が，$T(n) = 2n + n\log_2 n$ とする．
  $n \geqq 4$ なるすべての n に対して

$$
\begin{aligned}
T(n) &= 2n + n\log_2 n \\
&= n(2 + \log_2 n) \\
&\leqq n(\log_2 n + \log_2 n) \\
&= 2n\log_2 n
\end{aligned}
$$

が成り立つので $T(n)$ が $O(n\log_2 n)$ とわかる

19

19

計算量(2)

- 時間計算量（time complexity）
  - アルゴリズムの実行にどれだけ時間がかかるか
- 領域計算量（space complexity）
  - アルゴリズムの実行にどれだけ領域が必要か
- 時間と空間のトレードオフを考えるには，
  時間計算量と領域計算量の兼ね合いを考慮
- 単に「計算量」なら前者（時間）

20

20

計算量(3)

- 最大計算量（worst-case complexity）
  - 最悪の入力データを想定した計算量
- 平均計算量（expected complexity）
  - すべての入力データに対する計算量の平均
- 単に「計算量」といえば前者（最大）
- ある種のアルゴリズムでは平均計算量が重要
  - 例：クイックソート
    - 平均計算量：$O(n \log n)$
    - 最大計算量：$O(n^2)$
    - アルゴリズムの工夫で実質的には $O(n \log n)$ で実行可能

21

21

具体例:探索の計算量

- n 個のデータが登録されているテーブルから
  特定のキーを持つデータを探し出す処理
- 探索1回当たりにかかる時間はテーブルに
  登録されたデータの個数によって変化
- 計算量を求める際にはデータの個数を入力の
  大きさ n とする
- 線形探索法（linear search）
- 二分探索法（binary search）

注：詳しい説明については教科書（pp.12～27）を精読すること

22

22

探索に対する2つのpublicメソッド

- addメソッド：
  引数としてキー（int型）と値（Object型）
  を受け取りテーブルに追加．返り値はなし．
- searchメソッド：
  引数としてキー（int型）を受け取り，キー
  に対応した値を返す．見つからなければnull
  を返す．

23

23

線形探索法(1)(searchメソッド)

```
Entry[] table = new Entry[MAX]; // データを格納する配列
int n = 0;                      // 登録されているデータの個数

public Object search(int key)
{
  int i = 0;                    // (1)
  while (i < n) {               // (2)
    if (table[i].key == key)    // (3)
      return (table[i].data);   // (4) 見つかった
    i++;                        // (5)
  }
  return null;                  // (6) 見つからなかった
}
```

24

24

線形探索法(2)

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int i = 0;                        // (1)
  while (i < n) {                   // (2)
    if (table[i].key == key)        // (3)
      return (table[i].data);       // (4)
    i++;                            // (5)
  }
  return null;                      // (6)
}
```

1. (1)を1度実行
2. (2)～(5)のwhileループを (3)が成立するまで実行
3. (3)が成立すると(4)を1度実行
4. データが見つからなかった場合は (6)を1度実行

i = 0　i = 1　i = 2　i = 3　i = 4

配列table の添字	0	1	2	3	4
key	1	10	2	4	6
data	One	Ten	Two	Four	Six

25

25

オーダーの計算

- $T_1(n) = O(f(n))$ ， $T_2(n) = O(g(n))$ ならば
  加算：$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
  乗算：$T_1(n)\,T_2(n) = O(f(n)\,g(n))$
  が成立する
- 特に $O(1)$ は定数オーダー（constant order）と呼ばれ，n に独立なある定数に抑えられることを意味する

増加率

小 ← → 大

1　$\log n$　n　$n \log n$　n^2　n^3　……　n^k　2^n

26

26

線形探索法(3)

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int i = 0;                        // (1)
  while (i < n) {                   // (2)
    if (table[i].key == key)        // (3)
      return (table[i].data);       // (4)
    i++;                            // (5)
  }
  return null;                      // (6)
}
```

ステートメント	実行回数	計算量
(1)	1	$O(1)$
(2)	$n/2$	$O(n)$
(3)	$n/2$	$O(n)$
(4)	1	$O(1)$
(5)	$n/2$	$O(n)$
(6)	1	$O(1)$

- 各ステートメントのオーダーは $O(1)$
- ループは平均して $n/2$ 回

計算量 = $O(1) + O(n) + O(1) + O(1)$
= $O(\max(1, n, 1, 1))$
= $O(n)$

27

27

二分探索法(1)(searchメソッド)

```
Entry[] table = new Entry[MAX]; // データを格納する配列
int n = 0;                      // 登録されているデータの個数

                                前提条件：データはキーの昇順に整列済み
                                線形探索法とは異なる
public Object search(int key)
{
  int low = 0;                                  // (1)
  int high = n - 1;                             // (2)
  while (low <= high) {                         // (3)
    int middle = (low + high) / 2;              // (4)
    if (key == table[middle].key)               // (5)
      return table[middle].data;                // (6) 見つかった
    else if (key < table[middle].key)           // (7)
      high = middle - 1;                        // (8)
    else  // key > table[middle].key である
      low = middle + 1;                         // (9)
  }
  return null;                                  // (10) 見つからなかった
}
```

28

28

二分探索法(2)-1

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int low = 0;                                  // (1)
  int high = n - 1;                             // (2)
  while (low <= high) {                         // (3)
    int middle = (low + high) / 2;              // (4)
    if (key == table[middle].key)               // (5)
      return table[middle].data;                // (6)
    else if (key < table[middle].key)           // (7)
      high = middle - 1;                        // (8)
    else  // key > table[middle].key である
      low = middle + 1;                         // (9)
  }
  return null;                                  // (10)
}
```

	key	
low →	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
middle →	13	table[4] < 14
	14	table[5]
	18	table[6]
	20	table[7]
	21	table[8]
high →	25	table[9]

key = 14 のデータを二分探索法で探す
low = 0, high = 9, middle = (0+9) / 2 = 4
着目範囲を後半へ

29

29

二分探索法(2)-2

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int low = 0;                                  // (1)
  int high = n - 1;                             // (2)
  while (low <= high) {                         // (3)
    int middle = (low + high) / 2;              // (4)
    if (key == table[middle].key)               // (5)
      return table[middle].data;                // (6)
    else if (key < table[middle].key)           // (7)
      high = middle - 1;                        // (8)
    else  // key > table[middle].key である
      low = middle + 1;                         // (9)
  }
  return null;                                  // (10)
}
```

	key	
	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
	13	table[4]
low →	14	table[5]
	18	table[6]
middle →	20	table[7] > 14
	21	table[8]
high →	25	table[9]

low = 5, high = 9, middle = (5+9) / 2 = 7
着目範囲を前半へ

30

30

二分探索法(2)-3

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であるから
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

	key	
	1	table[0]
	3	table[1]
	4	table[2]
	8	table[3]
	13	table[4]
low, middle →	14	table[5] = 14
high →	18	table[6]
	20	table[7]
	21	table[8]
	25	table[9]

low = 5, high = 6, middle = (5+6) / 2 = 5

key = 14 と table[5] が一致 → データが見つかった

31

31

二分探索法(3)

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であるから
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

1. (1)(2)を1度実行
2. (3)〜(9)のループを実行
3. (5)が成立すると(6)を実行
4. データがない場合は (10)を1度実行

- 各ステートメントのオーダーは $O(1)$
- ループは何回実行される？
  - 着目される範囲が毎回半分に
  - 実行される回数は $\log^2 n$ 回程度に → $O(\log n)$

32

32

二分探索法(4)

```
Entry[] table = new Entry[MAX];
int n = 0;

public Object search(int key)
{
  int low = 0;                          // (1)
  int high = n - 1;                     // (2)
  while (low <= high) {                 // (3)
    int middle = (low + high) / 2;      // (4)
    if (key == table[middle].key)       // (5)
      return table[middle].data;        // (6)
    else if (key < table[middle].key)   // (7)
      high = middle - 1;                // (8)
    else  // key > table[middle].key であるから
      low = middle + 1;                 // (9)
  }
  return null;                          // (10)
}
```

ステートメント	実行回数	計算量
(1)(2)	1	$O(1)$
(3)〜(9)	$\log n$	$O(\log n)$
(10)	1	$O(1)$

計算量 = $O(1) + O(\log n) + O(1)$
= $O(\max(1, \log n, 1))$
= $O(\log n)$

33

33

1つのデータ登録に必要な計算量

- 線形探索法
  - 配列の末尾に要素を追加するだけ = $O(1)$
- 二分探索法
  - キーの昇順にデータを並べる必要 = $O(n)$
    - 挿入する位置を探す = $O(\log n)$
    - 後ろ側の平均して n/2 個のデータを移動 = $O(n)$
    - データを入れる = $O(1)$

34

34

線形探索法と二分探索法の比較

- データ登録と探索の計算量

手法	登録（n要素あたり）	探索（1回あたり）
線形探索法	$O(n)$	$O(n)$
二分探索法	$O(n^2)$	$O(\log n)$

- 二分探索法において，初期段階で全データを登録する場合は計算量の削減が可能
  - 登録時にキーの順番を無視
  - 登録後に $O(n \log n)$ の整列を実行
  - 結果，トータルの計算量は $O(n \log n)$ に

35

35

その他の探索手法

- ハッシュ法
  - キーの値を配列の添字へと変換する関数（ハッシュ関数）を利用した高速探索アルゴリズム
  - 詳細は教科書の第8章にて
  - 計算量は登録も探索も $O(1)$
  - データの個数よりも大きめの配列が必要

36

36

アルゴリズムを選択する基準(1)

- 計算量で評価すると
  ハッシュ法 ＜ 二分探索法 ＜ 線形探索法
- 他の性質
  - 線形探索法はデータを登録した順番が保存される
  - 二分探索法，ハッシュ法は保存されない
  - ハッシュ法では登録するデータよりも大きな配列が必要．ハッシュ関数によっては大幅な性能低下

37

37

アルゴリズムを選択する基準(2)

- n が小さいときはオーダーの大小よりも，定数係数の大小が支配的
  - オーダーの小さいアルゴリズムは「凝った」ものが多く，定数係数が大きくなる傾向
- プログラミングの手間
  - オーダーの小さいアルゴリズムは「凝った」ものが多いため，作成コスト大
- 時間と空間のトレードオフ
  - 空間を犠牲にしてオーダーを下げるアルゴリズム

38

38

教科書 第3章（pp.31～76）

データ構造とは？

詳細については，教科書を読むこと
Javaでの参照型の扱いについては
第2週の演習で実際に取り組むこと

39

39

プログラムを書くには

- 大雑把にスケッチ
  - 自然言語とプログラミング言語による疑似コード
- 段階的詳細化（stepwise refinement）
  - 必要なデータとデータに加える操作（アルゴリズム）が明確に
  - 操作を効率よく実現するためのデータの表現法（データ構造）が決まる

40

40

抽象データ型(abstract data type)

- データの型とその型に対する一連の操作の組
  - 例：整数の集合と和集合，差集合，積集合をとる操作の組
  - データの表現方法や操作の実現方法は規定しない
- 抽象データ型：辞書（探索の例）
  - データを登録
  - キーを指定してデータを検索
  - データを削除
  - すべてのデータを取得

41

41

カプセル化(encapsulation)

- データとそのデータに対する操作手続きを組みにすること
- 用意された操作を利用しなければデータを参照・変更できなくなる
  - プログラム開発や保守が容易に
  - 信頼性が向上

42

42

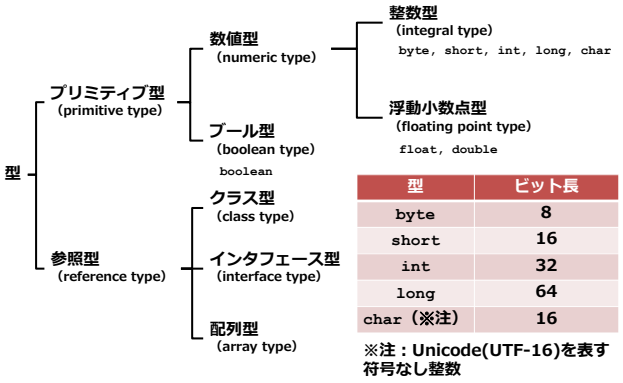
オブジェクト指向プログラミング

- Object-oriented programming
- カプセル化はOOPの特徴の１つ
  - データはオブジェクトの中に格納される
  - メソッドを使ってデータを操作できる
- OOPであるJavaは抽象データ型と親和性高

43

43

Javaにおけるデータ型



型	ビット長
byte	8
short	16
int	32
long	64
char（※注）	16

※注：Unicode(UTF-16)を表す符号なし整数

44

44

文字列の扱い

- Javaでは文字列はプリミティブ型ではない
- String型という参照型
- オブジェクトと同様の扱い
  - 文字列比較は==演算子ではなくequalsメソッド

45

45

ラッパークラス(wrapper class)

- プリミティブ型をオブジェクトとして扱う

プリミティブ型	ラッパークラス	値を取り出すメソッド
byte	Byte	byteValue
short	Short	shortValue
int	Integer	intValue
long	Long	longValue
char	Character	charValue
float	Float	floatValue
double	Double	doubleValue
boolean	Boolean	booleanValue

46

46

数値型とラッパークラスの変換

- ボクシング変換（boxing conversion）
  - 数値型をラッパークラスに変換
- アンボクシング変換（unboxing conversion）
  - ラッパークラスを数値型に変換
- オートボクシング／オートアンボクシング
  - Java5以降では自動的に変換
  - 便利だが変換にコストがかかる点に注意

```
Integer x = 100;
int a = x;
x = x + 20;
```

⇒

```
Integer x = new Integer(100);
int a = x.intValue();
x = new Integer(x.intValue() + 20);
```

47

47

ブール型(boolean)

- 論理値を表すデータ型
  - true（真）とfalse（偽）の2値のみ
- 比較演算子はboolean型の値を返す
- 論理演算子はboolean型を受け取り，boolean型の結果を返す
- 条件式にはboolean型が得られる式が必要

48

48

参照型

- クラス型（class type）
- インタフェース型（interface type）
- 配列型（array type）
- 参照型の変数にはデータを指す参照（reference）が入っている
  - 厳密には参照値（reference value）
- データ構造において他のデータを「指す」役割を持つ参照をリンク（link）と呼ぶ．リンクは参照だと考えること

49

49

オブジェクトのコピー

- PositionクラスとRobotクラスによる例
  - 第2週のプログラミング演習で扱います！
- ここでは別の例で説明

50

50

オブジェクトの生成

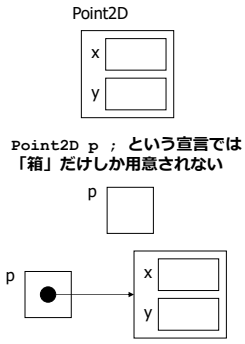
- クラス型を使ったコード

```
class Point2D {
  double x ;
  double y ;
}
```

```
Point2D p ;
```

```
Point2D p ;

p = new Point2D() ;
```



new してはじめてオブジェクトの領域が確保される

51

51

プリミティブ型と同じ？

- 変数をコピーした際に違いが発生

ソースコード

```
Point2D p1 ;
Point2D p2 ;

p1 = new Point2D() ;
p2 = new Point2D() ;

p1.x = 10.0 ;
p1.y = 20.0 ;

p2 = p1 ;   // ここで変数をコピー

p1.x = 15.0 ;

System.out.println("p2.x .." + p2.x) ;
```

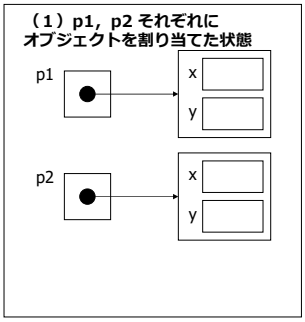
実行結果

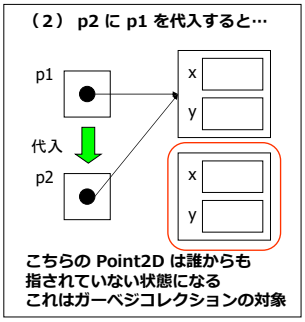
```
p2.x ..15.0
```

52

52

指し示す値だけをコピーした例





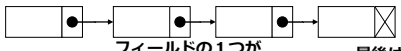
「Point2D を指し示すための値を格納する箱」であって「Point2D を格納できる箱」ではない

53

53

空参照(null reference)

- 「何も指していない状態」は空参照
- 具体例：連結リスト



フィールドの１つが
次のオブジェクトを指す
参照になっている

最後はnullにしておく

```
Point2D p ;

p = null ;

System.out.println(p.x) ;
```

空参照を無理に参照すると
実行時例外が発生

```
Exception in thread "main"
java.lang.NullPointerException
  at Null.main(Null.java:7)
```

54

54

オブジェクトのコピー(1)

- クラス型の変数は参照のため変数だけ代入してもオブジェクトそのものはコピーされない
- コピーするには新しいオブジェクトを new して内容を書き写す

```
Point2D p1 ;
  :
Point2D p2 = new Point() ;
p2.x = p1.x ;
p2.y = p1.y ;
```

- 面倒
- private なフィールドは setter がなければコピーできない

55

55

オブジェクトのコピー(2)

- カプセル化：オブジェクト指向の概念のひとつ
  - クラスの内部情報はクラスの中に隠蔽し，外部からは公開されたインタフェース（メソッド）だけで操作する
- 「自分自身をコピーする」という仕事は自分自身にさせるべき

```
class Point2D {
  double x ;
  double y ;

  public Point2D copy() {
    Point2D newPoint = new Point2D() ;

    newPoint.x = this.x ;
    newPoint.y = this.y ;

    return (newPoint) ;
  }
}
```

```
p2 = p1.copy() ;
```

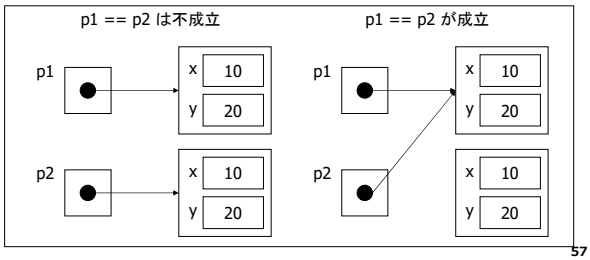
Point2D を複製したい側では，copy() メソッドを呼ぶ

56

56

オブジェクトの比較

- クラス型のオブジェクトを比較するには？
  - Point2D p1，p2 が等しいかどうかを調べたい
  - クラス型は所詮参照だから p1 == p2 で比較すると…
  - 必要であれば内容を比較する equals メソッドを実装



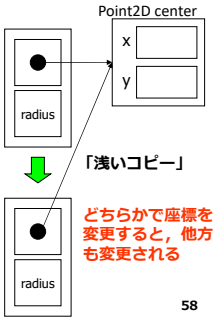
57

57

続･オブジェクトのコピー(1)

- 参照を含むオブジェクトのコピー
- 単純にフィールドを書き移してコピー
  - 浅いコピー（shallow copy）

```
class Circle {
  Point2D center ;
  double radius ;
    :
  public Circle copy() {
    Circle newCircle = new Circle() ;
    newCircle.center = this.center ;
    newCircle.radius = this.radius ;
  }
}
```

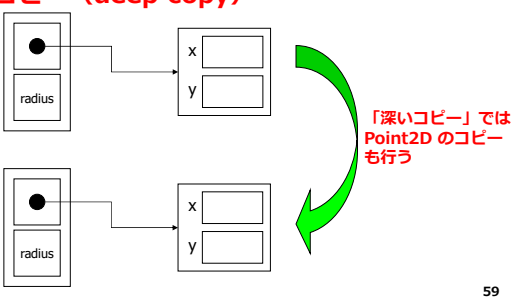


58

58

続･オブジェクトのコピー(2)

- フィールドの指す先にあるオブジェクト（例ではPoint2D）もコピーする
  - 深いコピー（deep copy）



59

59

その他の事項

- 配列
  - Javaにおける配列の扱い：参照型
- 列挙型
  - 特殊な種類のクラスとして実装
- ジェネリック型
  - Java5以降に導入された機能
  - 型パラメータを受け取るクラスを定義可能

Java言語の仕様なので詳細説明は割愛

60

60

まとめ

- アルゴリズムとは？
  - アルゴリズム＋データ構造＝プログラム
- 計算量
  - 時間計算量・領域計算量，最大計算量・平均計算量
  - オーダー記法
- データ構造とは？
  - 抽象データ型
  - カプセル化
  - オブジェクトのコピー

61

61

参考文献

- 定本 Javaプログラマのための アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3 アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造 Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇 オングス (著)・杉山 貴章・後藤 大地 (監修)
- Java 謎＋落とし穴 徹底解明 前橋和弥 著，技術評論社
- エッセンシャルJava 2nd Edition 宮坂雅輝 著，ソフトバンクパブリッシング
- 改訂版 Java言語 プログラミングレッスン上・下 結城浩 著，ソフトバンクパブリッシング

62

62