PRG1 (7): オブジェクト，クラス，メモリ管理

脇田建

2016.10.24

Scalaのデータ

- 原子的データ
  - Bool, Byte, Short, Int, Long, Float, Double, String, Symbol
- 関数：T1 → T2
- 構造的データ – 部分から構成されている.
  - 項：(T1, T2, …)
  - リスト：List[T]
  - 配列：Array[T]

構造的データ – コンストラクタ

- 項：v1:T1, v2:T2, … ならば **(v1, v2, …)** : (T1, T2, …)
- リスト
  - Nil
  - v: T かつ l: List[T] ならば **v::l** : List[T]

    3::(2::(1::Nil)) : List[Int]　　**“abc”::(“def”::Nil)** : List[String]
- 配列：v1: T, v2: T, … ならば Array(v1, v2, …) : Array[T]

  Array(1, 2, 3) : Array[Int]　Array(“abc”, “def”) : Array[String]

構造的データ – 部分の抽出

- 項：パターンマッチ
  - (1, 2, 3) match {
    case (x, y, z) => x + y + z } → 6
- リスト：パターンマッチ
  - **def** len[T](list: List[T]): Int =
    list **match** {
    case Nil => 0
    case _ :: list => 1 + len(list)
    }
- 配列：演算子
  - **val** v = Array(1, 2, 3)
  - **v(1)** → 2

代入

- 項へは代入不可
- リストへは代入不可
- 配列
  - val v = Array(1, 2, 3)
    v(1) → 2
    v(1) = v(1) + 1
    v(1) → 3

走査：列の走査

- 要素の走査

- val v = List(1, 2, 3)
  for (x <- v) println(x)

- val v = Array(1, 2, 3)
  for (x <- v) println(x)

走査：インデックスを用いた配列の走査

- インデックスを用いた走査
  - **for (i <- v.indices)** println(f"v($i) = ${v(i)}”)
  - ※ Array(3, 3, 3, 3, 3).indices ⇒ Range(0, 1, 2, 3, 4)

寄り道：添字に興味がないなら、関数的にすっきりと書ける

- **v.foreach**(x -> println(x)) あるいは v.foreach(println)

- **v.map**((x:Int) -> x*x*x)

- Listについても使えます

  - Range(1, 10).toList.map((x:Int) => x * x * x)

寄り道：Scalaの項についての注意

- scala> ()
- scala> (0)
  res39: Int = 0
- scala> (0, 1)
  res40: (Int, Int) = (0,1)
- scala> (0, 1, 2)
  res41: (Int, Int, Int) = (0,1,2)
- scala> (0, 1, 2, 3)
  res42: (Int, Int, Int, Int) = (0,1,2,3)

寄り道：項の要素数は22個までらしい

- scala> (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
  res44: (Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int) = (0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0)

- scala> (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1)
  res45: (Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int, Int) = (0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1)

- scala> (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2)
  <console>:8: **error: object <none> is not a member of package scala**
  (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2)
  ^

寄り道：Scalaの項の裏側を覗くと

package scalaのAPIマニュアルより

```
trait Product20[+T1, +T2, +T3, +T4, +T5, +T6, +T7, +T8, +T9, +T10, +T11, +T12, +T13, +T14, +T15,
+T16, +T17, +T18, +T19, +T20] extends Product
```

Product20 is a cartesian product of 20 components.

```
trait Product21[+T1, +T2, +T3, +T4, +T5, +T6, +T7, +T8, +T9, +T10, +T11, +T12, +T13, +T14, +T15,
+T16, +T17, +T18, +T19, +T20, +T21] extends Product
```

Product21 is a cartesian product of 21 components.

```
trait Product22[+T1, +T2, +T3, +T4, +T5, +T6, +T7, +T8, +T9, +T10, +T11, +T12, +T13, +T14, +T15,
+T16, +T17, +T18, +T19, +T20, +T21, +T22] extends Product
```

Product22 is a cartesi

```
trait Product3[+T1, +T
```

Product3 is a cartesia

```
trait Product4[+T1, +T
```

Product4 is a cartesian product of 4 components.

Product22 までしか
定義されてない

巨大な構造をどのように扱う？

現実の問題（スマホ@kakaku.com）

1. 製品名
2. 人気
3. レビュー評価
4. クチコミ件数
5. 登録日
6. キャリア
7. SIMサイズ
8. OS種類
9. 販売時期
10. 画面サイズ
11. 内蔵メモリ
12. テザリング
13. バッテリー
14. カラー
15. 最大待受時間
16. CPUコア数
17. 文字入力方法
18. 幅
19. 高さ
20. 厚み
21. 重量
22. 画面解像度
23. パネル種類
24. Wi-Fi
25. Bluetooth
26. NFC
27. カメラ画素数
28. 撮影用フラッシュ
29. 手ぶれ補正
30. サブカメラ
31. 4K撮影
32. GPS
33. 海外使用

Nexus 6を項で表すと...

- val nexus6 = (“Nexus 6 32GB SIMフリー”, 66, 0, 224, “2014/11/13”, “ワイモバイル”, “USIMカード(nanoSIM)”, ‘Android, “?”, 5.96, 64, true, …)

巨大な構造の扱い

- 部分構造が同じ型の場合
  - アクセスの順序が線形なら List[T]
  - ランダムアクセスが必要なら Array[T]
- さまざまな型のデータを複合したい場合
  - 巨大な項： (v1, v2, v3, …)

巨大な項の扱いの問題

- 項が内蔵できる項目数は22まで
- 多数の項目を持つ項に対するパターンマッチは困難
- 10番目の要素を取り出すために，こんなことは書きたくない

```
t = (v1, v2, …, v20)
t match {
    case (x1: T1, x2: T2, …, x20: v20) => { x10 }
}
```

ひとつの解決

- 巨大な項を分解したものを項としてまとめる
- val t = ((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12))
- t match { case (_, (_, _, y3, _), _) => y3 }　→ 7

ひとつの解決

- 巨大な項を分解したものを項としてまとめる
  - val t = ((1, 2, 3, 4), (5, 6, 7, 8), (9, 10, 11, 12))
  - t match { case (_, (_, _, y3, _), _) => y3 }　→ 7
- もうひとつの問題
  - 項目数が増えると順番を覚えられない

オブジェクト

- object nexus6 {
  val name = “Nexus 6 32GB SIMフリー”

  val popularity = 66
  val nReviews = 0
  val nComments = 224
  val since = “2014/11/13”
  val carrier = “ワイモバイル”

  val simSize = ‘nanoSIM
  val osType = ‘Android
  val available = “?”
  val displaySize = 5.96
  val memoryGB = 64
  val tethering = True
  …
}

- object iphone6 {
  val name = “iPhone 6 64GB SIMフリー”

  val popularity = 49
  val nReviews = 4.13
  val nComments = 451
  val since = “2014/9/10”
  val carrier = “SIMフリー”

  val simSize = ‘nanoSIM
  val osType = ‘iOS
  val available = “?”
  val displaySize = 4.7
  val memoryGB = 64
  val tethering = Hopefully
  …
}

nexus6オブジェクト vs iphoneオブジェクト

- print(nexus6.name)　→ “Nexus 6 32GB SIMフリー”
- print(iphone6.name)　→ “iPhone 6 64GB SIMフリー”
- for (phone <- List(nexus6, iphone6))
  print(phone.name)

<console>:12: error: value **name** is **not a member of Object**
for (phone <- List(nexus6, iphone6)) print(phone.name)
^

nexus6とiphone6の型は？

- scala> nexus6
- res0: **object_bug.nexus6.type** = object_bug.nexus6$@44df3cac
- scala> iphone6
  res8: **object_bug.iphone6.type** =
  object_bug.iphone6$@5945d85b
- scala> val phones = List(nexus6, iphone6)
  phones: **List[Object]** = List(object_bug.nexus6$@44df3cac,
  object_bug.iphone6$@5945d85b)

List(nexus6, iphone6)の要素にアクセスできない

- scala> val aPhone = phones(0)
  aPhone: Object = object_bug.nexus6$@44df3cac

- scala> nexus6.name
  res3: String = Nexus 6 32GB SIMフリー

- scala> aPhone.name
  <console>:17: error: value name is not a member of Object
  aPhone.name
  ^

オブジェクトは便利だが，そのままでは集合的に使えない。そこで、traitあるいはクラス

- クラスはオブジェクトの雛形：クラスを雛形にオブジェクトを製造

- クラス：オブジェクト

- 鯛焼器：鯛焼

クラスの定義

- class クラス名(クラスの引数) {
    val 定数名 = …
    var 変数名 = …
    def 関数の定義
}

オブジェクトの作成

- new クラス名(クラスの実引数)

Complexクラス

- class Complex(_re: Double, imaginary: Double) {

```
    val re = _re
    val im = imaginary
    def plus(c: Complex): Complex = {
      new Complex(_re + c.re, imaginary + c.im)
    }
    def minus(c: Complex): Complex = {
      new Complex(re - c.re, im - c.im)
    }
    ...
}
```

クラス，オブジェクト，インスタンス

- Complexクラスを雛形として，Complex型のオブジェクトを作成する.

- Complexをnewして，Complexクラスのインスタンスを得る

- 複素数 3+4i を表すオブジェクト c は，Complexクラスのインスタンスです.

- 複素数 3+4i を表すオブジェクト c のクラスは Complex です.

newとメモリ管理

- new Complex(re, im) を実行すると，Complexクラスのインスタンスが生成される＝このインスタンスを管理するための記憶領域が消費される.

- 例: complex1.scala: Complexを1M個生成したときの記憶領域の消費状況を調査するコード

memory.scala: 記憶領域消費量の計測

- (1 << 20, すなわち1M個)のComplexクラスのインスタンスを生成したときの記憶領域消費量を計測

- runtime.gc(): その時点以後に利用されないとわかっている記憶領域を掃除(garbage collect → gc)して，将来再利用できるようにとっておく.

- memoryTest1とmemoryTest2の微妙な違いが，大きな計測結果の違いをもたらすことに注意

結果の分析: memoryTest1

- 要素数Nの配列vを作成する．しかし，Scalaの実行時システムはvが以後の計算で用いられないことを発見し，直後の runtime.gc で配列が占めている記憶領域を回収する．このため，あたかも配列が記憶領域を占めていないかのような結果が表示された．

```
[info] Test 1> 0 bytes used.  0 bytes, or 0 words, per a Complex object
```

結果の分析: memoryTest2

- Scalaの実行時システムに配列vが後の計算で使用されることを知らせるために，配列の要素を乱択してメソッドの返り値としている.

- Complexオブジェクト1個について6ワード.

```
[info] Test 2> 54525968 bytes used.  52 bytes, or 6 words, per a Complex object
```

  - Arrayの要素としてComplexオブジェクトを指すポインタ (1word)

  - Complexオブジェクトの生成パラメタ(_re, _im: 2words)

  - val re; val im: 2words

  - 残り1wordは，オブジェクトとクラスの関係を表現するためのオーバーヘッド

小さな修正で記憶領域を削減可能

- val 宣言を def 宣言に変更するだけ．つまり，re, im を定数ではなく，メソッドとして再定義

  val re = _re ⇒ def re = _re

  val im = _im ⇒ def im = _im

- 定数を記憶する必要がなくなり2ワード分削減

```
[info] Test 2> 37748752 bytes used.  36 bytes, or 4 words, per a Complex object
```

練習問題: complex.scalaの修正

- object Complex (class Complexではないことに注意)にplusメソッドを追加しなさい.
  - def plus(c1: Complex, c2: Complex, c3: Complex) {
    ...
    }
- plusの働きはc1とc2の和をc3に保存すること．このためには，まずは class Complexでval宣言されている re と im を変数 (var) として宣言しなおして下さい.
  - val c1 = new Complex(3, 0)
    val c2 = new Complex(0, 4)
    val c3 = new Complex(0, 0)
    Complex.plus(c1, c2, c3) → このあと c3 = 3 + 4i となる.
- plus ができたら，minus(c1, c2, c3), times(c1, c2, c3), neg(c1, c2)を追加しなさい．それぞれ，減算，乗算，符号反転

練習問題: colorメソッドの修正(1)

- colorメソッドのwhile文のなかで実施する複素演算をComplex objectに追加したメソッドを用いて書き直しなさい.

- ここでの変更が記憶領域無駄遣いの主因である．変更前と変更後で，それぞれMandelbrotを起動するときに表示されるログを比較し，その変化を論じなさい.

練習問題: colorメソッドの修正(2a)

- class Complex（object Classではないことに注意）に以下の修正を施しなさい.

- 定数として宣言されているreとimを変数にしなさい.

- def set(_re: Double, _im: Double) { ... }

- setの働きは，Complexオブジェクトのre, imをそれぞれ_re, _imで破壊的に変更すること.

練習問題: colorメソッドの修正(2b)

- colorメソッドの冒頭付近では，zの初期化にあたってnew Complex(...)している．これは，colorメソッドが呼び出されるたびに（つまりピクセル数だけ）実施されるため効率が悪い．

- zの宣言をcolorメソッドの外に移動しなさい

- colorメソッド内では，zを(0＋0i)に初期化したい．ついさっき，Complexクラスに追加した機能が使えるはずだ．

- この修正を施せば，一画面の描画ごとにピクセル数(800×600＝48万)個のComplexの生成を削減できる．

練習問題: colorメソッドの修正(2c)

- colorメソッド内では，zの初期化と同様にcも初期化している．このためにcolorメソッド内では明示的にnew Complexしていないが，実はcomplexメソッドのなかでnewしている．

- 元々の def complex(x: Double, y: Double): Complex = { ... }

- 変更後 def complex(x: Double, y: Double, c: Complex) { ... }

- この変更の気分は，object Complex の plus メソッドなどと同様

- complexメソッドの修正ができたら，それにあわせて color メソッド内の c の初期化を修正し，そのために new Complex が発生しないようにしなさい．

練習問題: colorメソッドの修正(2d)

- colorメソッドの最後に Color.hsb(30, 1, n / 256.0) を返しているが，このために Color.hsb メソッドは Color オブジェクトを生成してしまっている．つまり，ピクセルを点描するたびに Color オブジェクトを生成している．

- プログラムのなかで利用している色は，Color.hsb(30, 1, 0..255)の256種類しかない．あらかじめ，256色のカラーパレットを配列に用意し，Color.hsbメソッドを利用するかわりにカラーパレットに保存されている色を利用すればオブジェクトを生成せずにすむ．

- 以下の配列の宣言をcolorメソッドの外側に置き，Color.hsbの呼び出しをこの配列の参照で置き換えればよい

  - val colors: Array[Color] = Array.tabulate(256){ i => Color.hsb(30, 1, i / 256.0) }