

第 6 章 クラスとオブジェクト

Scala には組み込みの有理数型はありませんが、クラスを使って簡単に定義できます。次が実装例です。

```
class Rational(n: Int, d: Int) {
  private def gcd(x: Int, y: Int): Int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)
  val numer: Int = n/g
  val denom: Int = d/g
  def +(that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
      denom * that.denom)
  def -(that: Rational) =
    new Rational(numer * that.denom - that.numer * denom,
      denom * that.denom)
  def *(that: Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) =
    new Rational(numer * that.denom, denom * that.numer)
}
```

有理数を、分子 n と分母 d の 2 つのコンストラクタ引数をとるクラスとして定義しています。このクラスは有理数上の演算メソッドだけではなく、分子と分母を返すフィールドも提供します。各演算メソッドは操作の右オペランドをパラメータにとります。操作の左オペランドは、常にそのメソッドがメンバであるような有理数です。

プライベート・メンバ 有理数の実装では、2 つの整数の最大公約数を計算するプライベートなメソッド `gcd` を定義し、また、コンストラクタ引数の `gcd` を格納するプライベートなフィールド `g` も定義します。これらのメンバはクラス `Rational` の外からはアクセスできません。それらはクラスの実装において、コンストラクタ引数を公約数で割って、分子と分母を常に正規形(約分された形)にするために使います。

オブジェクト生成とアクセス 有理数の使い方の例として、 i の範囲が 1 から 10 まで時に、 $1/i$ の和を印刷するプログラムを示します。

```
var i = 1
var x = new Rational(0, 1)
while (i <= 10) {
  x += new Rational(1, i)
  i += 1
}
println(" + x.numer + "/" + x.denom)
```

`+` は左オペランドに文字列を、右オペランドに任意の型の値をとります。右オペランドを文字列に変換し、それを左オペランドに追加した結果を返します。

継承とオーバーライド Scala のすべてのクラスは親クラスを持ち、それを継承しています。クラス定義の中で親クラスが指定されていない場合、ルート型 `scala.AnyRef` が暗黙のうちに仮定されます (Java の実装でいうと、この型は `java.lang.Object` のエイリアスです)。たとえばクラス `Rational` を次のように定義しても同じことです。

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
}
```

クラスは親クラスのメンバをすべて継承します。継承したメンバを再定義 (つまり、**オーバーライド**) できます。たとえばクラス `java.lang.Object` は、オブジェクトの文字列表現を返すメソッド `toString` を定義しています。

```
class Object {
  ...
  def toString: String = ...
}
```

```
}
```

Object での toString は、オブジェクトのクラス名と数からなる文字列として実装されています。このメソッドを有理数のオブジェクト用に再定義すると役に立ちます。

```
class Rational(n: Int, d: Int) extends AnyRef {  
  ... // as before  
  override def toString = "" + numer + "/" + denom  
}
```

注意すべき点は、Java とは異なり、定義の再定義では手前に override 修飾子が必要なことです。

もしクラス A がクラス B を継承しているなら、型 A のオブジェクトは、型 B のオブジェクトが期待される所ではどこでも使用できます。このような場合、型 A は型 B に **適合する** (conform)、といいます。たとえば Rational は AnyRef に適合します。したがって Rational 値を AnyRef 型の変数に代入できます。

```
var x: AnyRef = new Rational(1, 2)
```

パラメータなしのメソッド Java とは異なり、Scala のメソッドはパラメータリストを必ずしも必要としません。次の square メソッドがその例です。このメソッドは単に名前を書くだけで呼び出されます。

```
class Rational(n: Int, d: Int) extends AnyRef {  
  ... // as before  
  def square = new Rational(numer*numer, denom*denom)  
}  
val r = new Rational(3, 4)  
println(r.square) // prints '9/16'*
```

これは、パラメータなしメソッドは numer のような値フィールドと同じようにアクセスされるということです。値とパラメータなしメソッドの違いはそれらの定義にあります。値の右辺はオブジェクトが作成された時に評価され、以後は値が変化しません。その一方で、パラメータなしメソッドの右辺はメソッドが呼ばれる度に評価されます。フィールドとパラメータなしメソッドのアクセス方法が統一されているのでクラス実装者の自由度が増します。しばしば、あるバージョンではフィールドだったものが次のバージョンでは計算された値だったりします。統一的なアクセスのおかげで、クライアントは変更による書き直しが不要になります。

抽象クラス 整数の集合に対して、2つの操作 incl と contains を持つクラスを書く課題について考えてみましょう。(s incl x) は集合 s の要素すべてと、要素 x を持つ新しい集合を返すべきです。(s contains x) は、集合 s が要素 x を含む時は true を、さもなければ false を返すべきです。そのような集合のインターフェイスは次のようになります。

```
abstract class IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
}
```

IntSet は **抽象クラス** と分類されます。これは2つの結果をもたらします。一つは、抽象クラスは **延期された** メンバを持ち、それらは宣言はあっても実装はありません。この場合、incl と contains の両方がそのようなメンバです。二つ目は、抽象クラスには未実装のメンバがあるかもしれないので、そのクラスのオブジェクトを new で生成できません。その一方で、抽象クラスは他のクラスの基底クラスとして使うことができ、継承したクラスでは延期されたメンバを実装します。

トレイト Scala では、abstract class の代わりにキーワード trait をよく使います。トレイトは、他のクラスに付け加えるための抽象クラスです。これはトレイトが未知の親クラスにメソッドやフィールドをいくつか追加するからです。たとえば、トレイト Bordered は様々なグラフィカルコンポーネントに縁を追加するのに使われるでしょう。別の使い方としては、様々なクラスによって提供される機能のシグネチャをトレイトが集約する、という Java のインターフェイスがするようなやり方です。

IntSet はこの分類に入るので、トレイトとしても定義できます。

```
trait IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
}
```

抽象クラスの実装 たとえば集合を二分木で実装することにしましょう。木の形には2つの可能性があります。空集合を表す木と、整数1つと2つの部分木からなる木です。次がその実装です。

```

class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}

```

EmptySet と NonEmptySet の両方とも IntSet を継承します。これは、型 EmptySet と NonEmptySet が型 IntSet に適合すること --- 型 EmptySet や NonEmptySet の値は、型 IntSet の値が必要なところではどこでも使えることを意味します。

演習 6.0.1 メソッド union と intersection を 2 つの集合の結びと交わりとなるように書きなさい。

演習 6.0.2 要素 x を取り除いた集合を返すメソッド

```
def excl(x: Int)
```

を追加しなさい。そうするために便利なテスト用メソッド

```
def isEmpty: Boolean
```

も、集合に対して実装しなさい。

動的束縛 (Scala を含む) オブジェクト指向言語はメソッド呼び出しのときに **動的ディスパッチ** を用います。これは、メソッド呼び出しで起動されるコードは、メソッドを含むオブジェクトの実行時型に依存するという事です。たとえば式 `s contains 7`、ただし `s` は宣言された型 `s: IntSet` の値だとします。もしそれが `EmptySet` の値なら、実行されるのはクラス `EmptySet` の `contains` の実装であり、`NonEmptySet` の値の場合も同様です。この振る舞いは評価の置き換えモデルの直接的な帰結です。たとえば

```

(new EmptySet).contains(7)
-> (クラス EmptySet の contains 本体で置き換えることで)
false

```

あるいは

```

new NonEmptySet(7, new EmptySet, new EmptySet).contains(1)
-> (クラス NonEmptySet の contains 本体で置き換えることで)
  if (1 < 7) new EmptySet contains 1
  else if (1 > 7) new EmptySet contains 1
  else true
-> (条件式を書き換えて)
  new EmptySet contains 1
-> (クラス EmptySet の contains 本体で置き換えることで)
  false .

```

動的メソッドディスパッチは高階関数呼び出しと似ています。どちらの場合にも、実行されるコードの正体は実行時にしか分かりません。この類似性は表面上のものにとどまりません。実際、Scala はすべての関数値をオブジェクトとして表現しています (8.6 節参照)。

オブジェクト 整数集合の以前の実装では、空集合は `new EmptySet` で表現されていました。つまり空集合値が必要になる度に、毎回新しいオブジェクトが生成されました。値 `empty` を一度だけ定義して、その値をすべての `new EmptySet` の出現の代わりに使えば、不必要なオブジェクト生成を避けることができます。たとえば

```
val EmptySetVal = new EmptySet
```

この方法の問題の一つは、このような値定義は Scala の正しいトップレベルでの定義ではない、ということです。これは別のクラスかオブジェクトの一部でなくてはなりません。またクラス `EmptySet` の定義は少しばかりやり過ぎています。もしたった一つのオブジェクトにだけ関心があるなら、なぜオブジェクト用のクラスを定義するのでしょうか？ より直接的な方法は **オブジェクト定義** を使うことです。代わりに、より洗練された空集合の定義は次のようになります。

```
object EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}
```

オブジェクト定義の構文はクラス定義の構文に従います。オプションの `extends` 節とオプションの本体があります。クラスの場合のように、`extends` 節はオブジェクトが継承するメンバを定義し、一方で本体はオーバーライドするあるいは新規のメンバを定義します。しかしオブジェクト定義はただ1つのオブジェクトだけを定義するため、同じ構造を持った他のオブジェクトを `new` で生成できません。したがってオブジェクト定義には、クラス定義にはあるかもしれないコンストラクタパラメータはありません。

オブジェクト定義は Scala プログラムのどこにでも、トップレベルにも、書けます。Scala ではトップレベルのエンティティの実行順序は固定されていないため、オブジェクト定義によって定義されたオブジェクトがいつ生成され初期化されるのか、正確に知りたい人もいるでしょう。その答えは、オブジェクトはそのメンバが最初にアクセスされる時に生成される、というものです。この戦略は **遅延評価** と呼ばれています。

標準クラス Scala は純粋なオブジェクト指向言語です。これは Scala のすべての値がオブジェクトである、ということです。実際 `int` や `boolean` のようなプリミティブ型でさえ特別扱いされません。それらはモジュール `Predef` において、Scala クラスの型エイリアスとして定義されています。

```
type boolean = scala.Boolean
type int = scala.Int
type long = scala.Long
...
```

コンパイラは効率化のために通常、`scala.Int` 型の値を 32 bit 整数で、`scala.Boolean` 型の値を Java の `boolean` で、等と表現します。しかし必要に応じてこれらの特別な表現をオブジェクトに変換します。たとえばプリミティブの `Int` 値が `AnyRef` 型のパラメータをとる関数に渡される時です。したがってプリミティブ値の特別な表現は単に最適化のためであり、プログラムの意味を変えません。

次がクラス `Boolean` の仕様です。

```
package scala
abstract class Boolean {
  def && (x: => Boolean): Boolean
  def || (x: => Boolean): Boolean
  def ! : Boolean
  def == (x: Boolean) : Boolean
  def != (x: Boolean) : Boolean
  def < (x: Boolean) : Boolean
  def > (x: Boolean) : Boolean
  def <= (x: Boolean) : Boolean
  def >= (x: Boolean) : Boolean
}
```

`Boolean` はクラスとオブジェクトだけを使って定義でき、組み込みのブール値や数値の型を参照する必要はありません。`Boolean` 型の一つの実装を次に示します。これは標準 Scala ライブラリの実際の実装ではありません。効率化のために標準の実装では、組み込みのブール値を使います。

```
package scala
abstract class Boolean {
  def ifThenElse(thenpart: => Boolean, elsepart: => Boolean)
  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def ! : Boolean = ifThenElse(false, true)
  def == (x: Boolean) : Boolean = ifThenElse(x, x.!)
  def != (x: Boolean) : Boolean = ifThenElse(x.!, x)
  def < (x: Boolean) : Boolean = ifThenElse(false, x)
  def > (x: Boolean) : Boolean = ifThenElse(x.!, false)
  def <= (x: Boolean) : Boolean = ifThenElse(x, true)
}
```

```

def >= (x: Boolean) : Boolean = ifThenElse(true, x.!)
}
case object True extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = t
}
case object False extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = e
}

```

次はクラス Int の仕様の一部です。

```

package scala
abstract class Int extends AnyVal {
  def toLong: Long
  def toFloat: Float
  def toDouble: Double
  def + (that: Double): Double
  def + (that: Float): Float
  def + (that: Long): Long
  def + (that: Int): Int // analogous for -, *, /, %
  def << (cnt: Int): Int // analogous for >>, >>>
  def & (that: Long): Long
  def & (that: Int): Int // analogous for |, ^
  def == (that: Double): Boolean
  def == (that: Float): Boolean
  def == (that: Long): Boolean // analogous for !=, <, >, <=, >=
}

```

クラス Int も原理的にはオブジェクトとクラスだけで、組み込みの整数型を参照することなく実装できます。その方法を知るために、少しでも簡単な問題、自然数を表す型 Nat の実装方法を考えましょう。次は抽象クラス Nat の定義です。

```

abstract class Nat {
  def isZero: Boolean
  def predecessor: Nat
  def successor: Nat
  def + (that: Nat): Nat
  def - (that: Nat): Nat
}

```

クラス Nat の操作を実装するために、子オブジェクト Zero と子クラス Succ (次の数) を定義します。それぞれの数 N は、Zero に Succ コンストラクタを N 回適用したものです。

```

new Succ( ... new Succ(Zero) ... )
<----- N 回 ----->

```

オブジェクト Zero の定義はそのままです。

```

object Zero extends Nat {
  def isZero: Boolean = true
  def predecessor: Nat = error("negative number")
  def successor: Nat = new Succ(Zero)
  def + (that: Nat): Nat = that
  def - (that: Nat): Nat = if (that.isZero) Zero
                           else error("negative number")
}

```

Zero における、前の数(predecessor)および減算(-)関数の実装は、Error 例外を送出し、与えられたエラーメッセージと共にプログラムをアボートします。

次は「次の数」クラスの実装です。

```

class Succ(x: Nat) extends Nat {
  def isZero: Boolean = false
}

```

```

def predecessor: Nat = x
def successor: Nat = new Succ(this)
def + (that: Nat): Nat = x + that.successor
def - (that: Nat): Nat = x - that.predecessor
}

```

メソッド `successor` の実装に注意して下さい。ある数の次の数を作るために、`Succ` コンストラクタにオブジェクト自身を引数として渡す必要があります。オブジェクト自身は予約語 `this` で参照できます。

`+` と `-` の実装はそれぞれ、コンストラクタ引数を受け手とする再帰呼び出しを含みます。再帰は、受け手が `Zero` オブジェクトの時(そのように数を構成したので、それが起きることは保証されています) に終わります。

演習 6.0.3 整数の実装 `Integer` を書きなさい。実装はクラス `Nat` の操作すべてをサポートし、加えて、次の2つのメソッドもサポートしなさい。

```

def isPositive: Boolean
def negate: Integer

```

最初のメソッドは数が正であれば `true` を返すものとします。二つ目のメソッドは数の符号を変えます。実装の中で `Scala` の標準の数値クラスを使ってはいけません (ヒント: `Integer` の実装方法は二つ可能です。一つは既存の `Nat` 実装を利用し、整数を自然数と符号で表すものです。あるいは3つの子クラスとして `Zero` を 0 に、`Succ` を正の数のため、`Pred` を負の数のために使い、既知の `Nat` 実装を `Integer` へ一般化できます)。

この章で紹介した構文

型 (Types)

```
Type = ... | ident
```

型は、クラスを表す任意の識別子です。

式 (Expressions)

```
Expr = ... | Expr '.' ident | 'new' Expr | 'this'
```

式は、オブジェクト生成、オブジェクト値を持つ式 `E` のメンバ `m` を選択する `E.m`、あるいは予約語の `this` です。

定義と宣言 (Definitions and Declarations)

```

Def          = FunDef | ValDef | ClassDef | TraitDef | ObjectDef
ClassDef     = ['abstract'] 'class' ident ['(' [Parameters] ')'] ['extends' Expr] ['{' {TemplateDef} '}']
TraitDef     = 'trait' ident ['extends' Expr] ['{' {TemplateDef} '}']
ObjectDef    = 'object' ident ['extends' Expr] ['{' {ObjectDef} '}']
TemplateDef  = [Modifier] (Def | Dcl)
ObjectDef    = [Modifier] Def
Modifier     = 'private' | 'override'
Dcl          = FunDcl | ValDcl
FunDcl       = 'def' ident {'(' [Parameters] ')'} ':' Type
ValDcl       = 'val' ident ':' Type

```

定義は次のようなクラス、トレイト、あるいはオブジェクト定義です。

```

class C(params) extends B { defs }
trait T extends B { defs }
object O extends B { defs }

```

クラス、トレイト、オブジェクト定義内の定義 `def` の手前に、修飾子 `private` あるいは `override` を置けます。

抽象クラスとトレイトに宣言を含めることもできます。それらは **延期された** 関数や値を型とともに導入しますが、実装は与えません。延期されたメンバは、抽象クラスやトレイトのオブジェクトを生成する前に、サブクラスで実装しておく必要があります。

[前ページ](#) [6章](#) [目次](#) [次ページ](#)

- お疲れさまです。「新規のmンバ」は「新規のメンバ」の typoかと・・・ -- ryugate (2008-05-23 00:18:04)

を反映、訳で抜けている文章を追加、行頭の+は~+にしないとイケないらしい・・・ -- asaq (2008-10-05 13:46:14)

名前:	<input type="text"/>
コメント:	<input type="text"/>

投稿