第1节 A specification for class Rational

1.什么是分数对象

分数对象,表示分子/分母,并且分母不是0,分数和浮点数是有区别的,因为浮点数是有预估的,因此分数更精准。

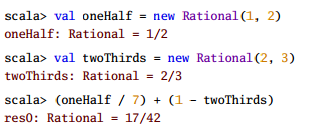
提供+ - \* /方法

+方法必须进行通分,然后分子和分母分别相加即可。

\*方法分子和分母进行相乘即可,然后进行约分.

/方法就是\*第二个数的倒数

比如



2.本节要构建一个不可变的分数对象,每次add或者乘法等等方法,产生一个新的分数对象,因此每一个原始的分数对象都是不可变的

第2节 Constructing a Rational 构建一个分数对象

1. class Rational(n: Int, d: Int) 客户端去构建一个分数对象,提供分子和分母

2.注意

scala> class Rational(n: Int, d: Int)

defined class Rational

scala> new Rational(5,7)

res12: Rational = Rational@1d4a3d2e

a.如果一个class没有任何body内容的时候,因此是可以省略{}的。

b.构造函数的时候会执行很多代码

scala的编译器可以编译在body中的任意代码,只要该代码不是field或者method,即在new的时候会执行非field和非method的代码块

class Rational(n: Int, d: Int) {

println("Created "+ n +"/"+ d)

}

打印结果

scala> new Rational(1, 2)

Created 1/2

res0: Rational = Rational@90110a

c. java与scala的不同在于,scala的构造函数中的参数,可以直接在body内使用,不需要像java一样,再次赋值才能使用

3.不可变对象编程的优点

a.不可变对象更容易编程,因为他们没有复杂的状态,不会随着时间变化

b.传递不可变对象非常自由,因为避免像可变对象一样要做一个副本,不可变对象不需要副本的概念

c.多线程变成的时候,更容易编程,因为没有线程可以改变该不可变对象的状态。

d.在hash中,不可变对象做key更安全,因为key要是变化了,可能就找不到原来存储的数据了

第3节 Reimplementing the toString method 重新实现toString方法

class Rational(n: Int, d: Int) {

override def toString = n +"/"+ d

}

第4节 Checking preconditions 校验前置条件

如何确保构造函数的数据的合法性

class Rational(n: Int, d: Int) {

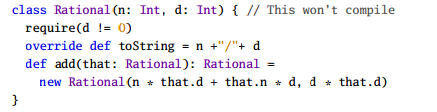
require(d != 0)

override def toString = n +"/"+ d

}

第5节 Adding fields 添加属性

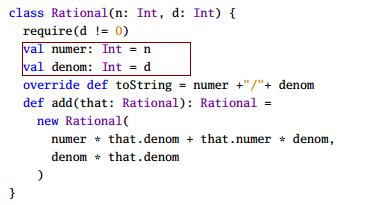
1.添加一个add方法,产生一个新的分数对象,分母是通分的结果,分子是通分后的数据加和



2.以上代码会编译出错

虽然该方法可以拿到n和d,但是却拿不到that的n和d,因此会出现编译异常。

3.修改代码



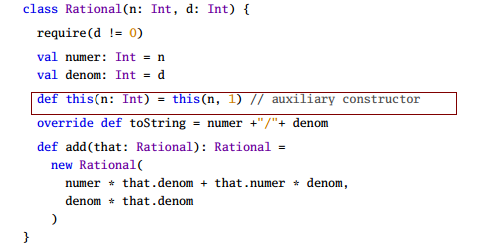
因为n和d只是该类的body可以看到,因此必须定义属性,该属性默认是public的,因此可以让其他类访问到,因此可以有that.number,以前是外部不能访问that.n的。

第6节 Self references 自引用

This表示该对象自己,因此使用this可以获取该对象本身的属性和方法,当然this也可以省略。

第7节 Auxiliary constructors 辅助构造函数

1. 有时候需要多个构造函数,scala中非主的构造函数都叫做辅助构造函数



调用val y = new Rational(3)

辅助构造函数会调用主构造函数

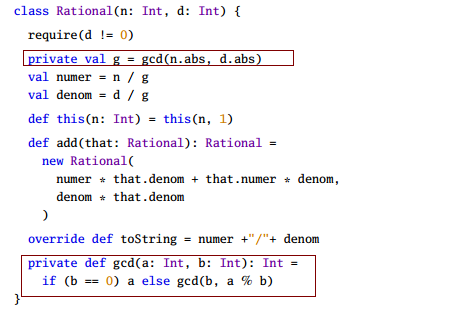
辅助函数以this作为方法名

2.scala和java构造函数的不同

java中构造函数第一个执行的必须是父类的super,或者执行其他的构造函数;scala中仅仅主函数能被执行,并且主函数才能执行super的构造器

第8节 Private fields and methods

1.为分数进行约分,分子和分母都除以最大公约数



定义了一个私有field和私有method,因为是私有的,因此只能class的body内可以访问该属性和方法,类外部是没办法访问的。

2. scala编译器会初始化所有的field,并且按照他们在原始文件中的出现顺序进行初始化,初始化的时刻就是在主构造函数构造过程中进行初始化的

3.整个class的初始化流程

class ConstructTest(n:Int,d:Int) {

println("init---")

val fieldInt = test()

def this(n: Int) = {

this(n, 1)

println("this2---"+fieldInt)

}

def test() : Int= {

println("test---")

5

}

}

object ConstructTest{

def main(args:Array[String]){

new ConstructTest(2)

}

}

输出

init---

test---

this---5

总结:

a.辅助函数第一行代码一定是调用主函数,因为在写其他方法时候都会报错。

b.所有代码都会先初始化该class,即主函数,然后按顺序初始化代码以及field,最后再执行辅助函数,即辅助函数内的代码,除了调用主函数的那行外,其他的都是在初始化真正的class后才会被调用的。因此辅助函数里面可以使用很多class的属性field了。

c.属性只是在初始化的时候调用一次,不会每次调用属性都去初始化,但是方法是每次调用都会被调用方法的逻辑,可以看到fieldInt使用两次,但是println("test---")只是在初始化的时候执行过一次而已。

第9节 Defining operators 定义操作符

1.想将add方法转换成+

2.即将add名字直接改成+即可,因此+和\*都是可以当作方法名字的

3.但是注意的是,+和\*等名字是有优先级规则的,因此当他们用做名字的时候,要特别注意

比如x + x \* y 的含义是 x + (x \* y),而不是(x + x) \* y:如果用+和\*符号代替文字的话,那么最好就要使用括号

第10节 Identifiers in Scala 在scala中的唯一标识符

1. alphanumeric(以\_和字母开头+字母/数字/\_) and operator这已经经常在scala中当作唯一标识符号

2.因为\_在scala中有特殊的意义,因此最好不要使用他命名,还是建议使用驼峰方式命名。

3.scala中常量也不要使用\_方式定义,建议也使用驼峰方式

4.操作符号

a.+, :, ?, ~ or #

比如scala中的操作符 + ++ ::: :->

b.scala其实是在内部将操作符进行了重新命名成合法的java名字.潜入了$字符,比如 :->的名字是$colon$minus$greater.

因此如果你想要使用java代码访问该操作符,则可以使用java的名字

c. scala与java的不同是,scala可以让identifiers变成任意长,没有限制

比如x<-y在java中被解析成4个字符,与 x < - y相同,但是在scala中<-被解析成单独的identifier

4. 一种混合的定义identifier方式.比如unary\_+,

5. scala中任意字符串,被``包裹起来的都可以当作identifier看待,比如`x` `<clinit>` `yield`

此时的名字在运行期的时候会取消``,包裹的内容做为identifier.

这种方式的主要目的是名字是关键字的时候,可以这么使用,比如java的Thread中静态方法yield,所以你不能写Thread.yield() ,因为yield在scala中是关键字,因此要想调用的话.则可以Thread.`yield`().

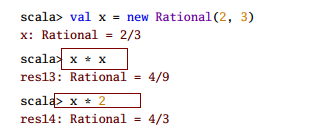
比如定义x方法

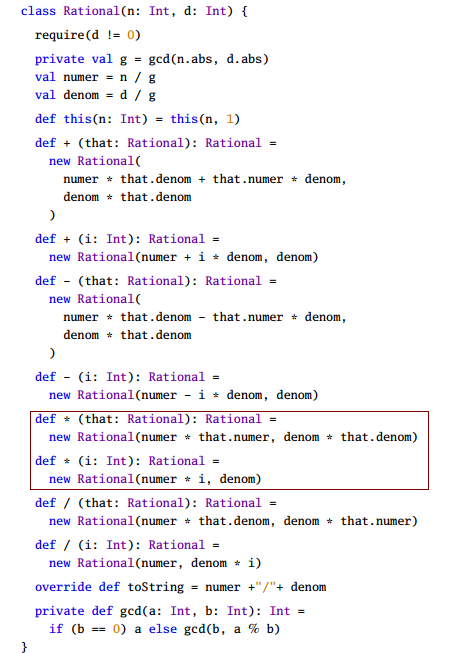
def `x` = {1}

x: Int

第11节 Method overloading

1．方法名字相同.参数不同,称作overloading,与java相同





第12节 Implicit conversions 隐式转换

1.我们以前执行的是r\*2,即r是分数对象,有\*方法,参数是2,但是我们反过来,执行2\*r的时候就报错,因为2\*r说明int有\*方法,参数是分数对象,这个很明显是不支持的。

2.可以隐式转换,自动将Int转换成分数对象,因此Int就有了分数对象对应的\*方法了

3.定义隐式转换

implicit def intToRational(x: Int) = new Rational(x)

a.implicit 表示告诉编译器这个方法是一个隐式转换方法

b.方法名字不重要,是为了可读性,给程序员看的

c.主要关注参数和返回值,即该隐式转换告诉编译器,将Int转换成Rational对象

d.隐式转换的作用域,在{}内可以被使用

e.隐式转换非常强大,但是也容易产生错误,因此详细的内容参见21章

第13节 A word of caution

主要提醒一下用户设计代码的时候,虽然+ \* 等操作符号以及隐式转换很强大,但是不要过度使用,避免带来代码不可读,不可维护的问题。