Chiselを始めたい人に読んでほしい本（七夕雅俊)_03章

Scalaの基本

HELLOWORLD

```
object HelloWorld {
  def main(args: Array[String]):Unit = println("HELLO WORLD!")
}
```

`object` で囲った中に `def main` を宣言する。

本には次のように記載あるが実際にはしたじゃないと動かない。なんだ？

```
bash $ sbt
.
.
.
sbt:chisel-modeule-template>
sbt:chisel-modeule-template> runMain main # こっちは動かなかった
sbt:chisel-modeule-template> runMain HellowWorld # こっちだと動いた
```

なんとなくだけど、runMainはオブジェクト中のエントリポイントであるmain関数をキックするためのもの、そのあとの引数はどのobjectを最初に読み込むかを指定するもののような気がする。どのオブジェクトがエントリポイントになるかに関しては設定見つからんかった。

sbtを使用したプロジェクトのディレクトリ例

```
./プロジェクトのルートディレクトリー
  |
  ├──── project : ビルドサポートファイル用ディレクトリー（Chiselでは気にしなくてOK）
  ├──── src/ : ソースコードを格納するディレクトリー
  |    |
  |    ├──── main
  |    |    |
  |    |    ├──── resources : メインに含むデータ（ChiselではRTLを置いたりする）
  |    |    └──── scala : メインのScalaソースファイル
  |    |
  |    └──── test
```

```
|         |
|         ├── resources : テストのデータファイルを置く
|         └── scala : テストのScalaソースファイル
|
└── build.sbt : sbtプロジェクトの設定ファイル
```

コマンド例

- compile
- run / runMain
- test / testOnly
- reload

sbtのコマンドはコマンド名の前に `~` をつけるとソースの変更が入るたびに自動的にそのコマンドを繰り返すようになる。デバッグとかに便利。

compile

`compile` でsbtがプロジェクトに含まれるScalaのソースファイルを自動的に検出してコンパイルを実行する。

```
// src/mainのソースのコンパイル
sbt:chisel-module-template> compile
// src/testのソースのコンパイル
sbt:chisel-module-template> test:compile
```

run/runMain

`run` コマンドはメイン関数を自動的に検出し、複数のメイン関数が存在する場合どれを実行するか尋ねる。 `runMain` コマンドは特定のmain関数を指定して実行する場合に使用する。

```
// run
sbt:chisel-module-template> run

// runMainは特定のメイン関数を指定して実行
sbt:chisel-module-template> runMain <実行したいメイン関数を含むオブジェクト名>

// src/test下のディレクトリーのメインを実行する場合は、頭にtest:を付与
sbt:chisel-module-template> test:runMain <実行したいメイン関数を含むオブジェクト名>
```

やっぱオブジェクト名必要じゃんか。上で実行できなかったのは本の記述とどこかに存在する？サンプルプログラムのオブジェクト名が食い違ってるからか。

test/testOnly

テストクラスに実装したテストを実行するためのコマンド。

```
// testはsrc/testの下のすべてのテストクラスを実行する
sbt:chisel-module-template> test

// testOnlyは指定したクラスのテストを実行する
sbt:chisel-module-template> testOnly <実行したいテストクラス>
```

reload

プロジェクトの再読み込み

```
sbt:chisel-module-template> reload
```

変数宣言と型

REPL

REPL(Read Eval Print Loop)はターミナル上でScalaのコードを対話的に実行して結果を確認できるScalaの機能。Scalaコマンドかsbtコマンドが実行可能な環境であれば使うことができる。

sbt shell上で `console` コマンドを実行すると対話実行形式のScalaシェルが実行できる。

```
sbt:chisel-module-template> console
[info] Starting scala interpreter...
Welcome to Scala 2.12.10 (OpenJDK 64-Bit Server VM, Java 11.0.14).
Type in expressions for evaluation. Or try :help.
scala> println("HELLO")
HELLO
```

val/var

- `var` 再代入可能
- `val` 再代入不可能

```
scala> var varA = 100
varA: Int = 100

scala> varA = 200
varA: Int = 200

scala> val valB = 100
valB: Int = 100
```

```
scala> valB = 200
<console>:12: error: reassignment to val
       valB = 200
```

関数型言語のため通常はvalをしようするケースがおおい（関数型言語よくわかってないのでこの意味がよく分からん）

Scalaは静的型付けを行い、変数への代入時に型推論が行われる。

```
scala> val test = true
test: Boolean = true

scala> test = 1 # testはboolean型なのでint型代入にはエラーをcallする。
<console>:12: error: reassignment to val
       test = 1
            ^

scala> val test = 1 # 再宣言すればOK。ポインタとかの概念あんのか？
test: Int = 1
```

整数型 Byte / Short / Int / Long / BigInt

1. Byte：符号付8bit整数
2. Short：符号付16bit整数
3. Int：符号付32bit整数
4. Long：符号付64bit整数。

いつもと一緒。

```
scala> val byteTest: Byte = 0 # 整数型を使用するときは(int以外)明確に型指定が必要。Longは？
byteTest: Byte = 0

scala> val byteTest2 = 0 # 型指定のない整数はint型と判定される。
byteTest2: Int = 0

scala>
scala> val byteVal: Byte = 128
<console>:11: error: type mismatch;
 found   : Int(128)
 required: Byte
```

れるエラーを直せば、ほぼコンパイルエラーが発生しない状態を作ることが可能です。

では改めて、Scalaの型について、REPLを使いながら確認してみましょう。

真理値型 - Boolean

まずはBoolean型です。真理値の真：true、もしくは偽：falseのどちらかを値として持ちます。

リスト3.14: Boolean型の宣言例

```
scala> val valTrue = true
valTrue: Boolean = true

scala> val valFlase = false
valFlase: Boolean = false
```

整数型 - Byte / Short / Int / Long / BigInt

次に整数型ですが、次の型が用意されています。

１．Byte ： 符号付き8bit整数

２．Short ： 符号付き16bit整数

３．Int ： 符号付き32bit整数

４．Long ： 符号付き64bit整数

"符号付き"と記載したとおりで、各整数型は最上位ビット（MSB）が符号ビットです。そのため、Chi

Byte型

Byte型は符号付きの8bit整数です。先に述べたように、Scalaは強力な型推論を備えているので、宣言

リスト3.15: Byte型の宣言例

```
scala> val byteVal: Byte = 0  // 左辺で型を明示
byteVal: Byte = 0

scala> val byteVal = 0.toByte // Int型のtoByteメソッドを使用
byteVal: Byte = 0

scala> val byteVal = 0:Byte   // 右辺で型を明示
byteVal: Byte = 0
```

Byte型は符号付きの8bitの整数なので、格納する値が符号付き8bitの値の範囲（-128～127）を超える

リスト3.16: Byte型の値の範囲

```
scala> val byteVal: Byte = 128
<console>:11: error: type mismatch;
 found   : Int(128)
 required: Byte
       val byteVal: Byte = 128      // 127を超えているのでエラー

scala> val byteVal: Byte = -129     // -128を超えているのでエラー
<console>:11: error: type mismatch;
 found   : Int(-129)
 required: Byte
       val byteVal: Byte = -129
scala> val shortVal: Short = 0     // 左辺で型を明示
shortVal: Short = 0

scala> val shortVal = 0:Short      // 右辺で型を明示
shortVal: Short = 0

scala> val shortVal = 0xf.toShort  // Int型のtoShortメソッドを使用
shortVal: Short = 0

scala> val shortVal: Short = 32768 // 32768を超えているのでエラー
<console>:11: error: type mismatch;
 found   : Int(32768)
 required: Short
       val shortVal: Short = 32768
scala> val intVal: Int = 0
intVal: Int = 0
<console>:1: error: integer number too large
       val intVal = 2147483648
scala> val intVal = 0:Int
intVal: Int = 0

scala> val intVal = 0.toInt
intVal: Int = 0

scala> val intVal = 2147483648
scala> val longVal:Long = 0
longVal: Long = 0
```

```
scala> val longVal = 0:Long
longVal: Long = 0

scala> val longVal = 0.toLong
longVal: Long = 0

scala> val longVal = 0L // Long型については"L"を付与しても宣言可能
longVal: Long = 0
```

BigInt

任意桁の整数を扱うための方。HW実装の際は64bitを超えるケースもよくあるので使うかも

```
scala> val bigIntVal: BigInt = 10
bigIntVal: BigInt = 10

scala> val bigIntVal = 10:BigInt
bigIntVal: BigInt = 10

scala> val bigIntVal = BigInt(10)
bigIntVal: scala.math.BigInt = 10
scala> val bigIntVal = BigInt(0xffffffff)
bigIntVal: scala.math.BigInt = -1          // BigInt型だけど-1になる
scala> val bigIntVal = BigInt("ffffffff", 16) // 16進数の文字列と基数を渡す
bigIntVal: scala.math.BigInt = 4294967295

scala> val bigIntVal = BigInt("4294967295", 10)
bigIntVal: scala.math.BigInt = 4294967295
```

Float/Donble

- Float：32bit浮動小数点型。
- Doubnle：16bit浮動小数点型。

型推論時はDouble型と推測される。

```
scala> val floatVal = 0.0f
floatVal: Float = 0.0

scala> val floatVal = 0.toFloat
floatVal: Float = 0.0
scala> val doubleVal = 0.0
```

```
doubleVal: Double = 0.0

scala> val doubleVal: Double = 0.0
doubleVal: Double = 0.0

scala> val doubleVal = 0.0:Double
doubleVal: Double = 0.0

scala> val doubleVal = 0.toDouble
doubleVal: Double = 0.0
```

Char/String

```
scala> val charVal: Char = 0x41  // 整数値を格納すると、対応する文字コードの文字になる
charVal: Char = A

scala> val charVal = 0x41.toChar // toCharを使っても同様
charVal: Char = A

scala> val charVal = 'a'         // ''の中の文字を入れると、その文字が格納される
charVal: Char = a

scala> val charA = 'あ'
charA: Char = あ

scala> val charA = '技'          // 漢字も使える
charA: Char = 技

scala> charA.toInt               // toIntを使用すると、文字コードが返ってくる
res0: Int = 25216

scala> val strA = "Chiselを始めたい人に読んで欲しい本"
strA: String = Chiselを始めたい人に読んで欲しい本

scala> strA(0)
res5: Char = C
```

コレクション

Cで言う配列、Pythonでいうリストやディクショナリに相当するデータ集合型

- Seq Chiselではモジュール定義でよく用いる
- Map ディクショナリ型のようにキーと値をペアで格納するRTL

```
scala> val seqA = Seq(0, 1, 3, 5, 7)
seqA: Seq[Int] = List(0, 1, 3, 5, 7)

scala> val seqB = Seq.fill(4)(0) // Seq.fill(<要素数>)(<埋めたい値>)
seqB: Seq[Int] = List(0, 0, 0, 0)

scala> val seqB = Seq.fill(4)(100)
seqB: Seq[Int] = List(100, 100, 100, 100)

scala> val c = seqB(0) // 要素にアクセスするときは"()"を使う
c: Int = 100

scala> val mapVal = Map(
     | "a" -> 100,
     | "b" -> 200
     | )
mapVal: scala.collection.immutable.Map[String,Int] = Map(a -> 100, b -> 200)
```

Unit

型を持たないことを表す型。Scalaは関数型言語なので必ず戻り値を持つが、 `println` のように戻り値を必要としない関数を表す値として定義され、こいつがreturnされる。(VoidとかNoneみたいなもんか。「Void」は無いわけだから厳密には違うかもだけど。。)

演算子

```
// 加減乗除は+, -, *, /を使用
scala> val a = 2 + 1
a: Int = 3

scala> val a = 2 - 1
a: Int = 1

scala> val a = 2 * 1
a: Int = 2
```

```
scala> val a = 2 / 1
a: Int = 2

// 剰余は%
scala> val a = 10 % 7
a: Int = 3

// 異なる型は演算が可能であればキャストされる
scala> val a = 2 / 1.0f
a: Float = 2.0

scala> val a = 2 / 1.0
a: Double = 2.0

scala> val a = 2 / 1.0f
a: Float = 2.0

scala> val a = 'a' + 10           // Char + Int
a: Int = 107

scala> val a: Char = 'a' + 10     // Char + Int
a: Char = k

scala> val a = 'a' - 'b'          // Char - Char
a: Int = -1

scala> val a: Char = '0' + '1'    // Char + Char
a: Char = a

scala> val a = 'a' * 'b'          // Char * Char
a: Int = 9506

scala> val a = 'a' / 'b'          // Char / Char
a: Int = 0

scala> val a = 'a' % 'b'          // Char % Char
a: Int = 97

scala> val a = "ab" + 'a'         // String + Char
a: String = aba
```

```
scala> val a = "ab" + "ab"        // String + String
a: String = abab

scala> val a = "ab" * 5           // String * Int
a: String = abababababab
```

ビット演算

Boolean型と整数型でのみ可能

```
scala> val a = 0x0f0f | 0xf0f0  // 論理和
a: Int = 65535

scala> val a = 0x0f0f & 0xf0f0  // 論理積
a: Int = 0

scala> val a = 0xffff ^ 0x0f0f  // 排他的論理和
a: Int = 65280

scala> val a = !true            // 否定：Boolean型："~true"にするとエラーが発生
a: Boolean = false

scala> val a = ~0xf0f0          // 否定：整数型："!0xf0f0"にするとエラーが発生
a: Int = -61681
```

比較演算

戻り値はBoolean型になる

```
scala> val a = true == true      // 等しい
a: Boolean = true

scala> val a = 99 != 100         // 等しくない
a: Boolean = true

scala> val a = 99 > 100          // 99は100より大きい
a: Boolean = false

scala> val a = 100 >= 100        // 100は100以上
a: Boolean = true
```

```
scala> val a = 99 < 100          // 99は100未満
a: Boolean = true

scala> val a = 99 <= 100         // 99は100以下
a: Boolean = true

scala> val a = 100
a: Int = 100

scala> val b = a > 99 && a < 101 // aは99より大きい、かつ、101より小さい
b: Boolean = true
```

制御構文

- if/match
- for/while

IF

```
scala> val a = 100
a: Int = 100

scala> :paste
// Entering paste mode (ctrl-D to finish)

if (a == 100)
  println("aは100")
else
  println("aは100でない")

// Exiting paste mode, now interpreting.

aは100

if (a != 10) {
  println("aは10でない")
} else if (a == 10) {
  println("aは10")
} else {
```

```
  println("aは何？？")
}

// Exiting paste mode, now interpreting.

aは10
a: Int = 10

scala> val a = 100
a: Int = 100

scala> val b = if (a == 100) a else 0  # IFは戻り値を持つ。
b: Int = 100
```

`:paste` は複数行エディットモードに入るREPLのコマンド。Scalaにおいての `{}` ブロック式ではブロック内の一番最後に評価された行が戻り値になる。ここでは `println` の戻り値がブロックの戻り値になる。

For

`i ← 0 until 10` で~9までのループ、 `i ← 0 to 10` で0~10までのループが実行される。 forの右側にはコレクションをとることもできる。

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

for (i <- 0 until 10) {
  println(i)
}

// Exiting paste mode, now interpreting.

scala> :paste
// Entering paste mode (ctrl-D to finish)

val s = Seq(0, 1, 2)
for (i <- s) {
  println(i)
}
```

```
// Exiting paste mode, now interpreting.

0
1
2
```

While

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

var i = 0
while (i < 3) {
  println(i)
  i += 1
}

// Exiting paste mode, now interpreting.
0
1
2

scala> :paste
// Entering paste mode (ctrl-D to finish)

var i = 0
do {
  println(i)
  i += 1
} while (i < 3)

// Exiting paste mode, now interpreting.

0
1
2
```

match

switchに似てるがもう少し柔軟な表現ができる。matchを使うことで柔軟なパラメタライズが可能となる。

```
<式> match {
  case <パターン> (if <ガード条件>) =>
    <式>
  ... // 以下caseを必要な分だけ実装
  case <パターン> (if <ガード条件>) =>
    <式>
}

scala> :paste
// Entering paste mode (ctrl-D to finish)

val a = 100
val b = if (a == 0) { 100 } else { 0 } match { // if式の戻り値は0なので
  case 0 => 50                                 // case 0にマッチしb = 50になる
  case _ => 99
}

// Exiting paste mode, now interpreting.

a: Int = 100
b: Int = 50
```

関数

Scala 2.xでは引数の数に制限があり、22個までとなっている。Scala 3.xではどうなってるのか要調査かな。
ブロック式は省略可能だが、書いたほうが無難な気がする。

```
// 関数定義
def <関数名>(<引数1>: <引数1の型>[, ... <引数N>: <引数Nの型>])[: <戻り値の型>] = {
  <関数の処理>
}

// 関数の呼び出し
関数名(引数1, 引数2)

scala> def add(a: Int, b: Int) = a + b
```

```
add: (a: Int, b: Int)Int

scala> def add(a: Int, b: Int): Int = {
     | a + b
     | }
add: (a: Int, b: Int)Int

scala> val c = add(10, 10)
c: Int = 20

// 複数の引数リストを持つ場合
scala> def add(a: Int)(b: Int): Int = a + b
add: (a: Int)(b: Int)Int

scala> val c = add(10)(10)
c: Int = 20

// デフォルト引数
scala> def add(a: Int = 10)(b: Int): Int = a + b
add: (a: Int)(b: Int)Int

scala> val c = add()(10)
c: Int = 20

// キーワード引数
scala> def add3(a: Int, b: Int = 10, c: Int): Int = a + b + c
add3: (a: Int, b: Int, c: Int)Int

scala> val c = add3(10, c = 10)
c: Int = 30
```

関数オブジェクト

関数は第１級オブジェクトなので関数をオブジェクトとして使用することが可能。ざっくりいうと関数を変数に格納できる。

```
scala> val add = (a: Int, b: Int) => a + b
add: (Int, Int) => Int = $$Lambda$4848/1157167742@dde0489
```

```
scala> val c = add(10, 10)
c: Int = 20
```

高階関数

ほかの関数をパラメタとして受け取ることや、戻り値として関数を返すことのできる関数。あまり自分で実装することは無いらしいが、コレクション型で定義済みの関数はパラメタライズを行う上で重要な要素となっている。

- map 元のコレクションたいしてなんらかの処理を行った結果を返す
- reduce/folt 元のコレクション型とその演算結果を使って処理を行うような場合。画像フィルターの処理などに使う。reduce/foldの違いは初期値を設定できるか否か
- filter 条件の指定した式にマッチした要素からなるリストを返却する。

```
// map
scala> val mapResult = Range(0, 10).map(_ * 2)
mapResult: scala.collection.immutable.IndexedSeq[Int] =
  Vector(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)

// reduce
scala> val reduceResult = Range(0, 10).reduce(_ + _)
recudeResult: Int = 45

// fold
scala> val foldResult = Range(0, 10).fold(5)(_ + _)
recudeResult: Int = 50

// filter
scala> val filterResult = Range(0, 10).filter(_ % 2 == 0)
recudeResult: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6, 8)

// foreach
scala> mapResult.foreach(println(_))
0
2
4
6
8
10
12
14
```

```
16
18

scala> val a = Range(0, 5).map(_ * 2).zipWithIndex
a: scala.collection.immutable.IndexedSeq[(Int, Int)] =
  Vector((0,0), (2,1), (4,2), (6,3), (8,4))
```

`_` は各コレクションの要素が格納されている。

クラス

```
class <クラス名>[<クラスパラメーター>] [extends <継承するクラス名> [with 継承するトレイト名]]
  <デフォルトコンストラクターの処理>

  [def <メソッド名>: <戻り値の型> = {

  }]
}
```

Chiselではデフォルトコンストラクター部分に回路記述を行う。宣言時に `val` をつけたパラメーターはpublic扱いとなる。

```
scala> class A
defined class A

scala> :paste
// Entering paste mode (ctrl-D to finish)

class A(val name: String)

// Exiting paste mode, now interpreting.
defined class A // class Aが定義された

scala> val instanceA = new A("diningyo")
instanceA: A = A@5d9764ae
```

クラスフィールド

フィールドの定義方法は次の2通り

1. <クラスパラメーター>部分で作成
    - ```
      scala> :paste
      // Entering paste mode (ctrl-D to finish)

      class A(val name: String)

      // Exiting paste mode, now interpreting.

      scala> val instanceA = new A("diningyo")
      instanceA: A = A@5d9764ae
      scala> println(instanceA.name) // nameフィールドを参照して値を表示する
      diningyo                       // 設定した値が表示された
      ```
2. <インスタンス時の処理>部分で作成
    - ```
      scala> :paste
      // Entering paste mode (ctrl-D to finish)

      class B (setName: String) {

        val name = setName // これもクラスのフィールド

        def getName(): String = name
      }



      // Exiting paste mode, now interpreting.

      defined class B

      scala> val instanceB = new B("diningyo")
      instanceB: B = B@19932c16

      scala> println(instanceB.name)
      diningyo

      scala> println(instanceB.setName)
      <console>:10: error: value setName is not a member of B
                    println(instanceB.setName)
      ```

```
scala> println(instanceB.getName)
diningyo
```

継承

単一検証のみ可能、多重継承は不可。 `trait` によるミックスイン(Mix-in)は可能。

```
class Base(val data: Int) {
  println(data)
}

class Derived(val data: Int) extends Base(data) {
  println(data * 2)
}
```

ChiselではHWモジュールはChiselの `Module` クラスを継承して実装する。 `abstract` キーワードで中小クラスの宣言が可能。（関数名とかだけ定義しておいて継承時にオーバーライド必須にさせるやつ。テンプレ的な）

```
scala> abstract class Base {
     |   val a: Int
     | }
defined class Base

//
scala> class Derived extends Base
<console>:12: error: class Derived needs to be abstract, since
 value a in class Base of type Int is not defined
       class Derived extends Base
```

ケースクラス

宣言時自動的にコンパニオンオブジェクトが生成されるため `new` によるインスタンス化が必要ない。モジュールに渡すクラス・パラメーターをまとめるためにけーす・クラスを用いることが多い。strictの強化版みたいなもんか。

```
case class <クラス名>[(<クラス・パラメーター>)][{
  <クラスの実装>
}]
scala> :paste
// Entering paste mode (ctrl-D to finish)

case class SampleCaseClass(
```

```
  a: Int,
  b: Boolean
)

// Exiting paste mode, now interpreting.

defined class SampleCaseClass

scala> val caseObj = SampleCaseClass(10, true)
caseObj: SampleCaseClass = SampleCaseClass(10,true)

// 自動的に宣言される
scala> caseObj.toString
res0: String = SampleCaseClass(10,true)

scala> val copyCaseObj = caseObj.copy(100)
copyCaseObj: SampleCaseClass = SampleCaseClass(100,true)
```

トレイト

Javaのインターフェースに似たもの。多重継承がサポートされていないため、複数のトレイトを合成するミックスインによって同様のことを行う。

```
trait <トレイト名> [extends <継承するクラス or トレイト>]
  [with <継承するクラス or トレイト>] {
  <トレイト内の記述>
}

scala> trait TraitA
defined trait TraitA
```

トレイト内の記述部分には以下のいずれかを定義可能

- 抽象フィールド ex.) val a :Int
- メソッド：中小メソッドでも実装を含んでもOK。

トレイトは単体ではインスタンスすることができず、クラスに継承する形で使用する。

```
scala> val a = new traitA
<console>:11: error: not found: type traitA
       val a = new traitA
```

```
scala> class ClassB extends TraitA // TraitAを継承
defined class ClassB

// トレイトのミックスイン
scala> trait TraitA
defined trait TraitA
scala> trait TraitB
defined trait TraitB

scala> class ClassC extends TraitA with TraitB
defined class ClassC

scala> val c = new ClassC
c: ClassC = ClassC@4d98974b
```

シングルトンオブジェクト

インスタンスが一つに限定されるオブジェクト。main関数。

コンパニオンオブジェクト

クラスと同じ名前のオブジェクトを作成できる。Chiselではコンパニオンオブジェクトを使用してハードウェア実装を行う。 `apply` というメソッドを実装すると `new` を使わずにインスタンスができる。
Javaで言うメソッドのオーバーロードみたいなもんか。
1行目と2行目の違いがよく分からん。上でも下でも行けるよってことか、

```
class A(val a: Int, val b: Int)

object A {
  def apply(a: Int): A = apply(a, 0)
  def apply(a: Int, b: Int): A = new A(a, b)
}
// コンパニオンオブジェクトのA.applyが呼び出される
scala> val b = A(100)
b: A = A@12f3f38

scala> println(b.a)
100
```

```
// どうやらクラスのインスタンスが自動作成されるらしい。かつapply句であれば引
scala> val b = A(100)
b: A = A@3d015dac

scala> val c = A(199, 200)
c: A = A@448a5d51

scala> println(c)
$line10.$read$$iw$$iw$A@448a5d51

scala> println(c.a)
199

scala> println(c.b)
200

scala> println(b.a)
100

scala> println(b.b)
0
```