Abstract

A member of a class or trait is abstract if the member does not have a complete definition in the
class. For instance, Java lets you declare
abstract classes and methods. But Scala goes
beyond that: you can declare abstract
fields and even abstract types as members of classes and traits.

Objectives

- Abstract members: vals, vars, methods, and types
- Pre-initialized fields
- Lazy vals
- Path-dependent types
- Enumerations

Part 1 :: Brief Review

The following trait declares one of each kind of abstract member: an abstract type (*T*), method(*transform*), val (*initial*), and var (*current*)

```
trait Abstract {
   type T
   def transform(x: T): T
   val initial: T
   var current: T
}
```

A concrete implementation of Abstract needs to fill in definitions for each of its abstract members. The implementation gives a concrete meaning to the type name *T* by defining it as an alias of type *String*.

```
class Concrete extends Abstract {
   type T = String
   def transform(x: String) = x + x
   val initial = "Hapiness"
   var current = initial
}
```

This is our first meeting with abstract elements. Next we explore details of the new forms of abstract members, as well as type members in general, are good for.

Part 2 :: Type Members

Type

The term *abstract type* in Scala means a
type declared to be a member of a class or trait, without specifying adefinition. An abstract type in Scala is commonly a member of some class or trait, such as type *T* in *trait Abstract*.

Of cause you can write such code in REPL (or scala worksheet)

```
type C = String
val c: C = "test string"
```

And the result will be

```
defined type alias C
c: C = test string
```

But it doesn't mean that abstract type can live its life in your architecture, it must be defined, at least in an arbitrary object. Otherwise it's useless.

It's convinient to think of non-abstract (concrete) type member such as type T in class Concrete, as a way to define a new name, or alias, for a type.

One reason to use a type member is to define a short, descriptive alias for a type whose real name is more verbose, or less obvious in meaning, than the alias. The other main use of type members is to declare abstract types that must be defined
in subclasses. It will be described later.

Vals

An abstract val declaration

```
val initial: String
```

The value can be given in certain implementation

```
val initial = "hi"
```

It's very similar as an abstract parameterless method declaration

```
def initial: String
```

What is the difference?

Client code would refer to both the val and the method in exactly the same way (i.e.,obj.initial). However, if initial is an abstract val, the client is guaranteed that obj.initial will yield the same value every time it is referenced. If initial were an abstract method, that guarantee would not hold because, in that case, initial could be implemented by a concrete method that returns a different value every time it's called.

So, any implementation of val must be a val definition; it may not be a var or a def.

```scala
abstract class Fruit {
    val v: String // `v' for value
    def m: String // `m' for method
}

abstract class Apple extends Fruit {
    val v: String
    val m: String // OK to override a \`def' with a \`val'
}

abstract class BadApple extends Fruit {
    def v: String // ERROR: cannot override a \`val' with a \`def'
    def m: String
}
```

Vars

```scala
trait AbstractTime {
    var hour: Int
    var minute: Int
}
```

If you declare an abstract var named hour, for example, you implicitly declare an abstract getter method, hour, and an abstract setter method, *hour_=*. There's no reassignable field to be defined—that will come in subclasses that define the concrete implementation of the abstract var.

```scala
trait AbstractTime {
   def hour: Int
   def hour_=(x:Int)
   def minute: Int
   def minute_=(x:Int)
}
```

Vals initialisation

Abstract vals sometimes play a role analogous to superclass parameters. This is particularly important for traits, because traits don't have a constructor to which you could pass parameters.

```scala
trait RationalTrait {
    val numerArg: Int
    val denomArg: Int
}
```

The RationalTrait trait given here defines instead two abstract vals: numerArg and denomArg. To instantiate a concrete instance of that trait, you need to implement the abstract val definitions.

```
new RationalTrait {
    val numerArg = 1
    val denomArg = 2
}
```

This expression yields an instance of an anonymous class that mixes in the trait and is defined by the body. This particular anonymous class instantiation has an effect analogous to the instance creation new Rational(1, 2).

The difference is in the order in which
expressions are initialized.

```
new RationalTrait {
    val numerArg = expr1
    val denomArg = expr2
}
```

expr1 and expr2, are evaluated as part of the initialization of the anonymous class, but the anonymous class is initialized after the RationalTrait. So the values of numerArgand denomArg are not available during the initialization of RationalTrait.

Is it a problem?

It becomes a problem in the variant ofRationalTrait shown in Listing 20.4, which defines normalized numerators and denominators.

```
trait RationalTrait {
   val numerArg: Int
   val denomArg: Int
   require(denomArg != 0)
   private val g = gcd(numerArg, denomArg)
   val numer = numerArg / g
   val denom = denomArg / g

   private def gcd(a: Int, b: Int): Int =
        if (b == 0) a else gcd(b, a % b)

   override def toString = numer \+ "/" \+ denom
}
```

If you try to instantiate this trait with some numerator and denominator expressions that are not simple literals, you'll get an exception:

```
scala> val x = 2
x: Int = 2
scala> new RationalTrait {
    val numerArg = 1 * x
    val denomArg = 2 * x
}
java.lang.IllegalArgumentException: requirement failed
at scala.Predef$.require(Predef.scala:207)
at RationalTrait$class.$init$(<console>:10)
... 28 elided
```

The exception in this example was thrown because denomArg still had its default value of 0 when class RationalTrait was initialized, which caused the require invocation to fail. This example demonstrates that initialization order is not the same for class parameters and abstract fields.

Pre-initializisaion

Pre-initialized fields in an object definition

```
scala> new {
    val numerArg = 1 * x
    val denomArg = 2 * x
} with RationalTrait
res1: RationalTrait = 1/2
```

Pre-initialized fields in a class definition.

```
class RationalClass(n: Int, d: Int) extends {
        val numerArg = n
        val denomArg = d
} with RationalTrait {
        def + (that: RationalClass) = new RationalClass(
                numer * that.denom + that.numer * denom,
   denom * that.denom
   )
}
```

Because pre-initialized fields are initialized before the superclass constructor is called, their initializers cannot refer to the object that's being constructed. Consequently, if such an initializer refers to this, the reference goes to the object containing the class or object that's being constructed, not the constructed object itself.

Lazy-vals

But it's more convinient to get system make a decision about order of evaluation. Lazy vals can help.

```
scala> object Demo {
val x = { println("initializing x"); "done" }
}
defined object Demo

scala> Demo
initializing x
res3: Demo.type = Demo$@2129a843
scala> Demo.x
res4: String = done

scala> object Demo {
lazy val x = { println("initializing x"); "done" }
}
defined object Demo
scala> Demo
res5: Demo.type = Demo$@5b1769c
scala> Demo.x
initializing x
res6: String = done
```

Initializing of Demo does not involve initializing x. The initialization of x will be deferred until the first time x is used.

This is similar to the situation where x is defined as a parameterless method, using a def. However, unlike a def, a lazy val is never evaluated more than once. In fact, after the first evaluation of a lazy val the result of the evaluation is stored, to be reused when the same val is used subsequently.

```
trait LazyRationalTrait {
   val numerArg: Int
   val denomArg: Int
   lazy val numer = numerArg / g
   lazy val denom = denomArg / g
   override def toString = numer \+ "/" \+ denom
   private lazy val g = {
          require(denomArg != 0)
          gcd(numerArg, denomArg)
        }
        private def gcd(a: Int, b: Int): Int =
            if (b == 0) a else gcd(b, a % b)
}
```

Sequence of initializations:

1. A fresh instance of LazyRationalTrait gets created and the initialization code ofLazyRationalTrait is run. This initialization code is empty; none of the fields ofLazyRationalTrait is initialized yet.
2. Next, the primary constructor of the anonymous subclass defined by the new expression is executed. This involves the initialization of numerArg with 2 and denomArg with 4.
3. Next, the toString method is invoked on the constructed object by the interpreter, so

that the resulting value can be printed.
4. Next, the numer field is accessed for the first time by the toString method in traitLazyRationalTrait, so its initializer is evaluated.
5. The initializer of numer accesses the private field, g, so g is evaluated next. This evaluation accesses numerArg and denomArg, which were defined in Step 2.
6. Next, the toString method accesses the value of denom, which causes denom's evaluation. The evaluation of denom accesses the values of denomArg and g. The initializer of the g field is not re-evaluated, because it was already evaluated in Step 5.
7. Finally, the result string "1/2" is constructed and printed.

Path-dependent types

```
class Food
abstract class Animal {
    type SuitableFood <: Food
    def eat(food: SuitableFood)
}

class Grass extends Food
class Cow extends Animal {
    type SuitableFood = Grass
    override def eat(food: Grass) = {}
}

class DogFood extends Food
class Dog extends Animal {
    type SuitableFood = DogFood
    override def eat(food: DogFood) = {}
}
```

```
scala> val bessy = new Cow
bessy: Cow = Cow@713e7e09

scala> val lassie = new Dog
lassie: Dog = Dog@6eaf2c57
scala> lassie eat (new bessy.SuitableFood)
<console>:16: error: type mismatch;
    found: Grass
    required: DogFood
        lassie eat (new bessy.SuitableFood)
```

The problem here is that the type of the SuitableFood object passed to the eat method,bessy.SuitableFood, is incompatible with the parameter type of eat, lassie.SuitableFood.

```
scala> val bootsie = new Dog
bootsie: Dog = Dog@13a7c48c
scala> lassie eat (new bootsie.SuitableFood)
```

Consider these two classes, Outer and Inner:

```
class Outer {
    class Inner
}
```

In Scala, the inner class is addressed using the expression Outer#Inner instead of Java'sOuter.Inner. The '.' syntax is reserved for objects. For example, imagine you instantiate two objects of type Outer, like this:

```
val o1 = new Outer
val o2 = new Outer
```

Here o1.Inner and o2.Inner are two path-dependent types (and they are different types). Both of these types conform to (are subtypes of) the more general type Outer#Inner, which represents the Inner class with an arbitrary outer object of type Outer. By contrast, typeo1.Inner refers to the Inner class with a specific outer object (the one referenced from o1). Likewise, type o2.Inner refers to the Inner class with a different, specific outer object (the one referenced from o2).

Refinement types

```
class Pasture {
    var animals: List[Animal { type SuitableFood = Grass }] = Nil
    // ...
}
```

Case study: Currencies

A typical instance of Currency would represent an amount of money in dollars, euros, yen, or some other currency. Should contain arithmetic operation.

A first (faulty) design of the Currency class

```
abstract class Currency {
    val amount: Long
    def designation: String
    override def toString = amount + " " + designation
    def + (that: Currency): Currency = ...
    def * (x: Double): Currency = ...
}

abstract class Dollar extends Currency {
    def designation = "USD"
}
abstract class Euro extends Currency {
    def designation = "Euro"
}
```

At first glance this looks reasonable. But it would let you add dollars to euros. The result of such an addition would be of type Currency. But it would be a funny currency that was made up of a mix of euros and dollars. What you want instead is a more specialized version of the +method.

A second (still imperfect) design of the Currency class

```
abstract class AbstractCurrency {
    type Currency <: AbstractCurrency
    val amount: Long
    def designation: String
    override def toString = amount + " " + designation
    def + (that: Currency): Currency = ...
    def * (x: Double): Currency = ...
}

abstract class Dollar extends AbstractCurrency {
    type Currency = Dollar
    def designation = "USD"
}
```

Operations

```
def + (that: Currency): Currency = new Currency {
    val amount = this.amount + that.amount
}
```

But it causes error

```
error: class type required
    def + (that: Currency): Currency = new Currency {
```

One of the restrictions of Scala's treatment of abstract types is that you can neither create an instance of an abstract type nor have an abstract type as a supertype of another class. So the compiler would refuse the example code here that attempted to instantiate Currency.

However, you can work around this restriction using a factory method.

```
abstract class AbstractCurrency {
    type Currency <: AbstractCurrency // abstract type
    def make(amount: Long): Currency // factory method
...
// rest of class
}
```

A design like this could be made to work, but it looks rather suspicious. Why place the factory method inside class AbstractCurrency?

```
abstract class CurrencyZone {
    type Currency <: AbstractCurrency
    def make(x: Long): Currency
    abstract class AbstractCurrency {
        val amount: Long
        def designation: String
        override def toString = amount + " " + designation
        def + (that: Currency): Currency =
            make(this.amount + that.amount)
        def * (x: Double): Currency =
            make((this.amount * x).toLong)
}
}
```

An example concrete CurrencyZone is the US, which could be defined as

```
object US extends CurrencyZone {
    abstract class Dollar extends AbstractCurrency {
        def designation = "USD"
}
    type Currency = Dollar
    def make(x: Long) = new Dollar { val amount = x }
}
```

This is a workable design. There are only a few refinements to be added. The first refinement concerns subunits. So far, every currency was measured in a single unit: dollars, euros, or yen. However, most currencies have subunits: For instance, in the US, it's dollars and cents.

```
class CurrencyZone {
...
    val CurrencyUnit: Currency
}
```

```
override def toString =
    ((amount.toDouble / CurrencyUnit.amount.toDouble)
    formatted ("%." + decimals(CurrencyUnit.amount) + "f")
    + " " + designation)

private def decimals(n: Long): Int =
    if (n == 1) 0 else 1 + decimals(n / 10)

def from(other: CurrencyZone#AbstractCurrency): Currency =
    make(math.round(
        other.amount.toDouble * Converter.exchangeRate
            (other.designation)(this.designation)))
```

Finally

```scala
abstract class CurrencyZone {
    type Currency <: AbstractCurrency
    def make(x: Long): Currency
    abstract class AbstractCurrency {
        val amount: Long
        def designation: String
        def + (that: Currency): Currency =
            make(this.amount + that.amount)
        def * (x: Double): Currency =
            make((this.amount * x).toLong)
        def - (that: Currency): Currency =
            make(this.amount - that.amount)
        def / (that: Double) =
            make((this.amount / that).toLong)
        def / (that: Currency) =
            this.amount.toDouble / that.amount
        def from(other: CurrencyZone#AbstractCurrency): Currency =
            make(math.round(
                other.amount.toDouble * Converter.exchangeRate
                    (other.designation)(this.designation)))
        private def decimals(n: Long): Int =
            if (n == 1) 0 else 1 + decimals(n / 10)
        override def toString =
            ((amount.toDouble / CurrencyUnit.amount.toDouble)
                formatted ("%." + decimals(CurrencyUnit.amount) + "f")
                + " " + designation)
    }
}
```