Software Composition Paradigms

Sommersemester 2015

Radu Muschevici

Software Engineering Group, Department of Computer Science

TECHNISCHE UNIVERSITÄT DARMSTADT

2015-05-05

Beyond Inheritance: Mixins & Traits

Mixins

Mixins

A mixin is a uniform extension of many different parent classes with the same set of fields and methods.

[Ancona, Lagorio, and Zucca 2000]

- Offers functionality to be inherited, but it cannot be instantiated – subclass without superclass – or with many superclasses
- Mixin can be reused with different superclasses.
- A class may inherit some or all if its functionality from one or more mixins via multiple inheritance.
- Mixins require certain functionality and provide additional functionality.
- Flexible and lots of opportunities for reuse.

[Bracha and Cook 1990]

~~Traits~~ Mixins in Scala

```scala
trait Singer {
  def sing { println( "singing ..." ) }
}
```

Trait defining some behaviour

```scala
class Insect
class Cicada extends Insect with Singer
class Bird extends Singer with Flyer
        // given Flyer as another trait
```

Static trait inheritance (builds new class)

```scala
class Person {
  def tell {
    println("here's a little story ...")
  }
}
val singingPerson = new Person with Singer
singingPerson.sing
```

Dynamic trait inheritance (anonymous class)

Mixins in Scala

- Scala calls its mixins “traits”.
- Scala traits are used to define object types by specifying the signature of the supported methods.
- Scala traits can be partially implemented, i.e. it is possible to define default implementations for some methods (unlike Java interfaces).
- Scala traits may not have constructors.
- Common uses: enriching thin interfaces and defining stackable modifications

[Odersky, Spoon, and Venners 2011]

Mixin with Requirements

```
abstract class WhoAmI {
    //Abstract class defines the required methods
    def whoami() : String
}
trait Singer extends WhoAmI {
    //Trait uses required method
    def sing { println("I am a singing " + whoami) }
}
class Insect extends WhoAmI {
    // Required method implementation
    def whoami = "insect"
}
class Cicada extends Insect with Singer

class Person extends WhoAmI {
    def whoami = "person"
    def tell { println("here's a little story about a "
                + whoami) }
}
```

Simple Example

```scala
trait Similarity {
    def isSimilar(x: Any): Boolean
    def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
```

- `isSimilar` is abstract
- `isNotSimilar` is concrete but written in terms of `isSimilar`
- Classes that integrate this trait only have to provide a concrete implementation for `isSimilar`. `isNotSimilar` gets inherited directly from the trait.

Simple Example (cont.)

```scala
class Point(x: Int, y: Int) extends Similarity {
    def isSimilar(obj: Any) =
        obj.isInstanceOf[Point] &&
        obj.asInstanceOf[Point].x == x
}

object TraitsTest extends Application {
    val p1 = new Point(2, 3)
    val p2 = new Point(2, 4)
    val p3 = new Point(3, 3)
    println(p1.isNotSimilar(p2)) // False
    println(p1.isNotSimilar(p3)) // True
    println(p1.isNotSimilar(2)) // True
}
```

The Ordered Trait

Thin vs. Rich Interfaces

- Rich interface: many methods (easier in theory for client)
- Thin interface: fewer methods – easier for implementer

```scala
trait Ordered[A] {
    def compare(that: A): Int
    def < (that: A): Boolean = (this compare that) > 0
    def > (that: A): Boolean = (this compare that) < 0
    def <= (that: A): Boolean = (this compare that) <= 0
    def >= (that: A): Boolean = (this compare that) >= 0
    def compareTo(that: A): Int = compare(that)
}
```

An `Ordered` interface should be rich (for convenience), i.e. supply all comparison operators: `<`, `>`, `<=`, `>=`.

The Ordered Trait (cont.)

An `Ordered` interface in Java would require that we implement all methods:

```
class Rational implements Ordered {
    boolean isLessThan(Rational that) {...}
    boolean isGreaterThan(Rational that) {...}
    boolean isLessOrEqualThan(Rational that) {...}
    boolean isGreaterOrEqualThan(Rational that) {...}
}
```

In Scala, implement only one method, `compare`, and get a rich interface:

```
class Rational (n : Int, d : Int) extends Ordered[Rational] {
    def compare (that: Rational) =
        (this.numer * that.denom) - (that.numer * this.denom)
}
```

Mixins as Stackable Modifications

- Use Scala traits to modify the methods of a class
- Stack these modifications onto each other

- Consider the `IntQueue` class

```scala
abstract class IntQueue {
    def get() : Int
    def put(x : Int)
}
```

- Now we'll build a concrete class atop of it

An Implementation

```scala
import scala.collection.mutable.ArrayBuffer

class BasicQueue extends IntQueue {
    private val buf = new ArrayBuffer[Int]
    def get() = buf.remove(0)
    def put(x : Int) { buf += x }
}
```

A Modification Trait

```
trait Doubling extends IntQueue {
    abstract override def put (x : Int) {
        super.put(2*x)
    }
}

val queue = new BasicQueue with Doubling
queue.put(10)
queue.get() // Result: 20
```

- Trait can only be mixed into `IntQueues`
- **super** refers to the class that actually uses the trait

Two additional, stackable traits

```
trait Incrementing extends IntQueue {
    abstract override def put (x : Int) { super.put(x+1) }
}
trait Filtering extends IntQueue {
    abstract override def put (x : Int) {
        if (x >= 0) super.put(x)
    }
}
```

```
val queue = new BasicQueue with Doubling with Filtering with Incrementing
queue.put(-1)
queue.get // Result: 0
```

```
val queue = new BasicQueue with Doubling with Incrementing with Filtering
queue.put(-1)
queue.get // Result: Exception (queue is empty)
```

The mixin order is significant!

Different mixin orders ⇒ different behaviours.

Linearisation (Method Resolution Order, MRO)

- Linearisation means that when a class is instantiated (**new**`...`), a linear order of its superclasses, traits and itself is determined (from most specific to least specific).
- When several methods are applicable for a given call, the one defined on the most specific class or trait, according to the linearisation, is selected.
- Linearisation resolves conflicts in method dispatch due to ambiguity.
- Linearisation is needed to provide a consistent dispatch order of a multiple inheritance hierarchy.

Desirable Properties of Linearisations

- Acceptable: linearisation depends only on shape of inheritance hierarchy

- Observes local precedence order: linearisation of a class is consistent with linearisation of superclasses. *If A precedes B for class C, then A will precede B for all subclasses of C.*

- Monotonicity: every property inherited by a class is define in or inherited by one of the direct super classes.

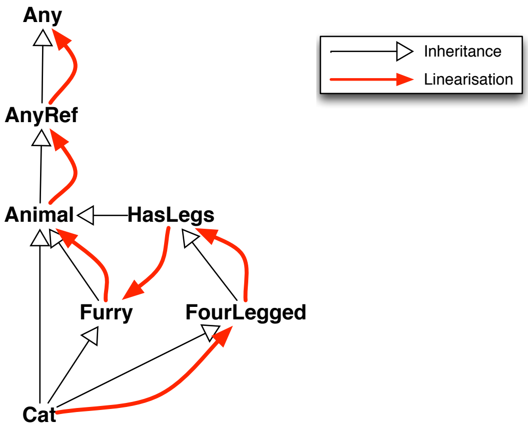
[Barrett et al. 1996]

Scala's Linearisation

1. Put the actual type of the instance as the first element.
2. Starting with the rightmost parent type and working left, compute the linearisation of each type, appending its linearisation to the cumulative linearisation. (Ignore `AnyRef` and `Any` for now.)
3. Working from left to right, remove any type if it appears again to the right of the current position.
4. Append `AnyRef` and `Any`.

Scala's Linearisation: Example

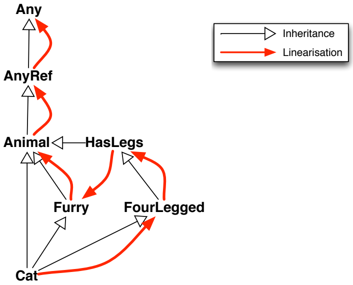
```scala
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```



Scala's Linearisation: Example (cont.)

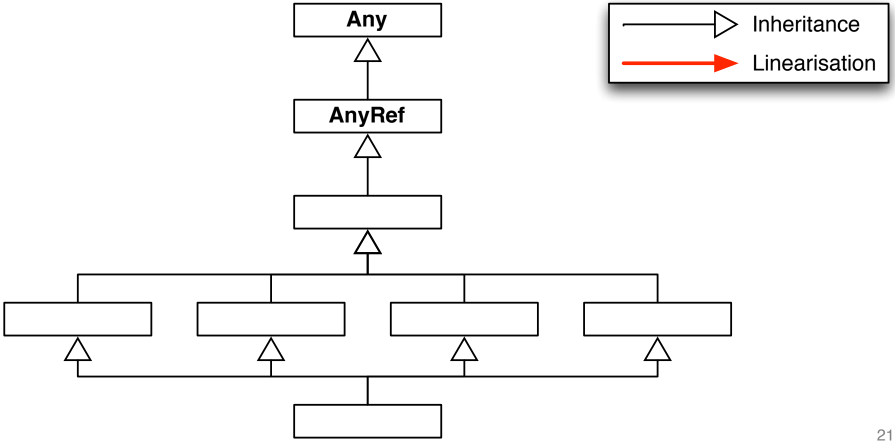
Apply rules 1–4 (slide 18):

1. `Cat`
2. `Cat` $\rightarrow$ `FourLegged` $\rightarrow$ `HasLegs` $\rightarrow$ `Animal` $\rightarrow$ `Furry` $\rightarrow$ `Animal` $\rightarrow$ `Animal`
3. `Cat` $\rightarrow$ `FourLegged` $\rightarrow$ `HasLegs` $\rightarrow$ ~~`Animal`~~ $\rightarrow$ `Furry` $\rightarrow$ ~~`Animal`~~ $\rightarrow$ `Animal`
4. `Cat` $\rightarrow$ `FourLegged` $\rightarrow$ `HasLegs` $\rightarrow$ `Furry` $\rightarrow$ `Animal` $\rightarrow$ `AnyRef` $\rightarrow$ `Any`



Scala Linearisation Exercise

```
abstract class IntQueue {...}
class BasicQueue extends IntQueue {...}
trait Doubling extends IntQueue {...}
trait Filtering extends IntQueue {...}
trait Incrementing extends IntQueue {...}
class MyQueue extends BasicQueue with Doubling with Filtering with Incrementing
```



Mixins vs. Multiple Inheritance

Mixins offer *late-binding* of **super** calls:

- With multiple inheritance the method called by **super** is determined statically.
- With mixins, the method is determined by *linearisation*, possibly at runtime.

Mixins vs. Multiple Inheritance: Example

Multiple inheritance (not Scala):

```
class A {
   read() : String {...}
   write(s : String) {...}
}
class SyncReadWrite {
   read() : String { ... super.read() ... }
   write(s : String) { ... super.write(s) ... }
}
class SyncA extends A, SyncReadWrite {...}
```

- Diamond problem: Which `read` and `write` methods will `SyncA` inherit?
- **super** calls refer to methods of the superclass of `SyncReadWrite`.

Mixins vs. Multiple Inheritance: Example (cont.)

Mixins:

```
class A {
    read() : String {...}
    write(s : String) {...}
}
trait SyncReadWrite {
    read() : String { ... super.read() ... }
    write(s : String) { ... super.write(s) ... }
}
class SyncA extends A with SyncReadWrite {...}
```

- Linearisation: `SyncA` $\rightarrow$ `SyncReadWrite` $\rightarrow$ `A`
- **super** calls refer to methods declared in `A`.

Problems with Mixins

Problem: little control over composition – a linearisation rule selects order of method calls.

- A suitable total ordering on features must be found.
- “Glue code” exploiting or adapting linear composition may be *dispersed* throughout the class hierarchy.
- Resulting class hierarchies are often *fragile* w.r.t. change – conceptually simple changes impact multiple parts of the hierarchy.

Traits

Traits: Motivation

- Inheritance: granularity too coarse – difficult to decompose application into an optimal class hierarchy, to maximises reuse.
- Mixins pose numerous problems for reuse, e.g. little control over composition.

Traits

A trait is *a set of methods* divorced from any class hierarchy.
[Ducasse et al. 2006]

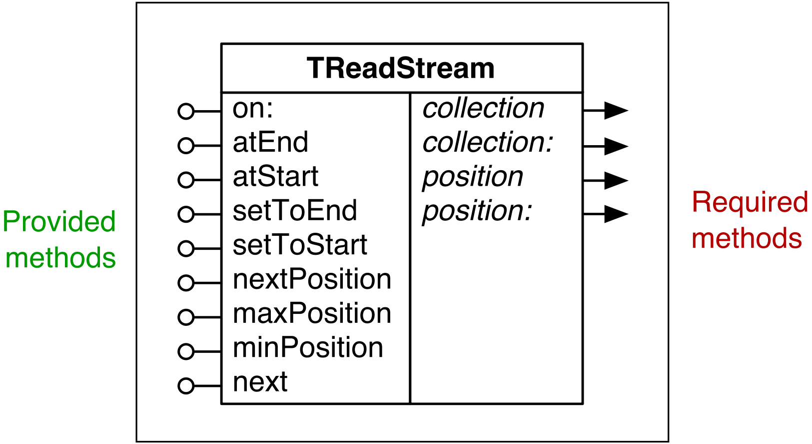
- Traits are purely units of reuse (only methods). Classes are generators of instances.
- Traits are simple software components that *provide* and *require* methods.
- Traits specify no state, so the only conflict when combining traits is a method conflict. Conflict resolution mechanisms are provided.

Traits

A trait is *a set of methods* divorced from any class hierarchy.
[Ducasse et al. 2006]

- Traits are purely units of reuse (only methods). Classes are generators of instances.
- Traits are simple software components that *provide* and *require* methods.
- Traits specify no state, so the only conflict when combining traits is a method conflict. Conflict resolution mechanisms are provided.
- Newer versions do specify state!

A Trait



[Ducasse et al. 2006]

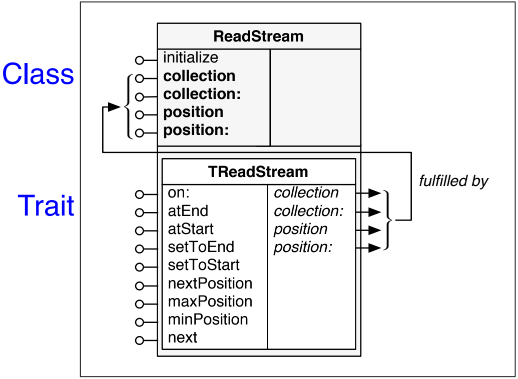
Trait Composition

- Classes are composed from traits: Incremental extensions to the *single* superclass are specified using one or more traits.
- Traits can be composed from other traits.
- Traits can be composed in arbitrary order.
- Principle: Composite retains complete control over composition.
  - Glue methods connect traits together, adapting provided trait methods, and resolving method conflicts.
  - Adaptation required only when conflicts are present or changes are required.

Trait Operations

- Composite traits are constructed using *trait sum*, *overriding*, *exclusion* and *aliasing*.
- Trait sum takes the *union* of the non-conflicting methods. Identical names that map to different method bodies are marked as a *conflict*.
- Overriding resolves conflicts by providing methods in the composite that replace trait methods with the same name.
- Exclusion avoid conflicts by excluding methods from composition.
- Aliasing renames methods, thus avoiding method name clashes.

Composition with Traits



Trait Composition Rule

- Methods defined in a class override methods provided by a trait. Methods in class implement *gluing* to resolve conflicts.

- Flattening property. A (non-overridden) method in a trait has the same semantics as if it were implemented directly in the class.

- Composition order is irrelevant. Disambiguation is explicit.

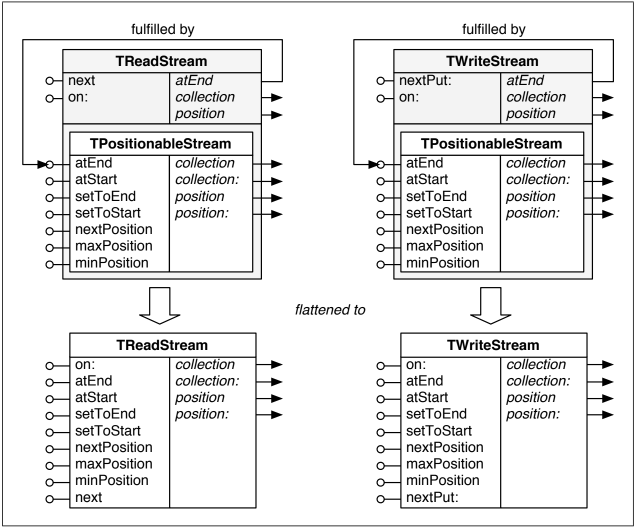
Flattening

In any class defined using traits, the traits can be *inlined* to obtain an equivalent class definition that does not use traits.

Traits can be compiled away.

- Methods defined in the class take precedence over methods provided by a trait.
- Trait methods take precedence over superclass methods.
- The keyword **super** has no special semantics for traits.

Flattening: Example



Conflict Resolution

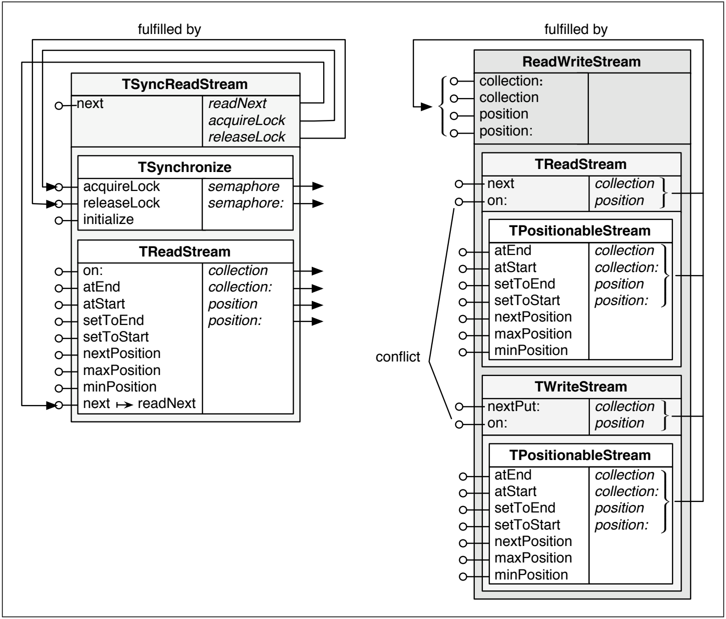
Conflicts

A conflict arises when composing two traits that provide identically named methods with different bodies.

Method conflicts are *resolved explicitly* by

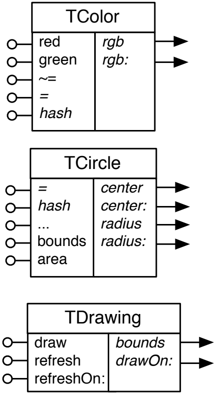
- Defining an overriding method in the composite class/trait,
- Exclusion in the composition clause,
- Aliasing, i.e. making a trait method available under another name.

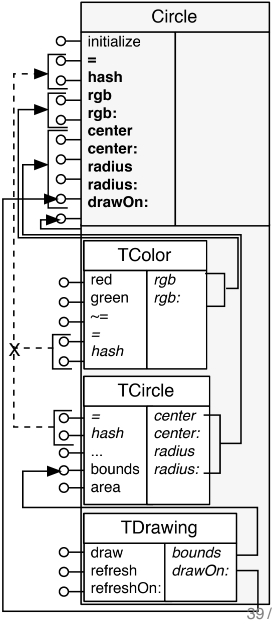
Conflict Resolution: Example



Exclusion Removes Conflicts

Exclude methods that would otherwise be provided by a trait in order to avoid a conflict.





Traits vs. Mixins

Mixins

1. A mixin is a unit of reuse, but (often) also defines a type
2. Composition by inheritance
3. Must be applied incrementally
4. Order of composition given by linearisation algorithm
5. Inheritance has two roles: code reuse and building conceptual hierarchies.

Traits

1. *Purely* units of reuse
2. Composition by sum, overriding, aliasing and exclusion: More fine-grained control
3. Several traits can be applied to a class in a single operation.
4. Composition is unordered & requires explicit resolution of conflicts.
5. Inheritance is separate from trait composition and only reflects conceptual hierarchies.

Stateful Traits
(next lesson)

This Week's Reading Assignment

- Bracha, G., and Cook, W. *Mixin-based inheritance*. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (New York, NY, USA, 1990), OOPSLA/ECOOP '90, ACM Press, pp. 303–311.

- Download link:
  `http://dl.acm.org/citation.cfm?id=97982`
- Freely accessible from within the TUD campus network

References I

Ancona, Davide, Giovanni Lagorio, and Elena Zucca (2000). "Jam – A Smooth Extension of Java with Mixins". In: *European Conference on Object-Oriented Programming*. Vol. 1850. LNCS. Springer, pp. 154–178.

Barrett, Kim, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington (1996). "A Monotonic Superclass Linearization for Dylan". In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA '96. ACM Press, pp. 69–82.

Bracha, Gilad and William Cook (1990). "Mixin-based inheritance". In: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA/ECOOP '90. ACM Press, pp. 303–311.

References II

Ducasse, Stéphane, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black (2006). "Traits: A Mechanism for Fine-grained Reuse". In: *ACM Transactions on Programming Languages and Systems* 28.2, pp. 331–388.

Odersky, Martin, Lex Spoon, and Bill Venners (2011). *Programming in Scala: A Comprehensive Step-by-Step Guide*. 2nd. USA: Artima Incorporation.