Traits in Scala

LAMP/EPFL

The Scala Programming Language

- A programming language designed for software components and component systems.
- Two claims:
  - A language for component software needs to be *scalable* in the sense that the same concepts can describe small as well as large parts.
    - Scalability can be achieved by a fusion of *object-oriented and functional programming.*
- Smalltalk goes in the same direction. Main difference is that Scala has a conventional syntax and is statically typed.
- For usability we aim for seamless integration with Java.

Scala in a Nutshell

- Scala has a uniform object model
  - all values are objects
  - all objects are instances of a class
  - all operations are method calls
- Functions are first-class values
- Uniform and powerful abstraction concepts for both types and values
- Specialization of classes and traits via single inheritance and a flexible mixin-composition mechanism
- External, retroactive extensibility via views
- Decomposition of objects using pattern-matching
- Lightweight support for XML
- Implemented on the Java and .Net platforms

Classes in Scala

- Scala features generic class abstractions that can be arbitrary nested:

```
class Buffer[T] {

  var xs: List[T] = Nil;
  def add(elem: T): unit = { xs = elem :: xs; }
  def elements: Iterator[T] = new BufferIterator

  class BufferIterator extends Iterator[T] {
    var ys = xs;
    def hasNext = !ys.isEmpty;
    def next = { ... }
  }

}
```

Single Inheritance

- Classes can be extended via single inheritance

```
class IterableBuffer[T] extends Buffer[T] {
  def forall(p: T => Boolean): Boolean = {
    val it = elements; var res = true;
    while (res && it.hasNext) { res = p(it.next); }
    res
  }
}
```

- Here is a second independent extension

```
class Stack[T] extends Buffer[T] {
  def push(elem: T): unit = add(elem);
  def pop: T = { val y = xs.head; xs = xs.tail; y }
}
```

Symmetric Mixin Composition

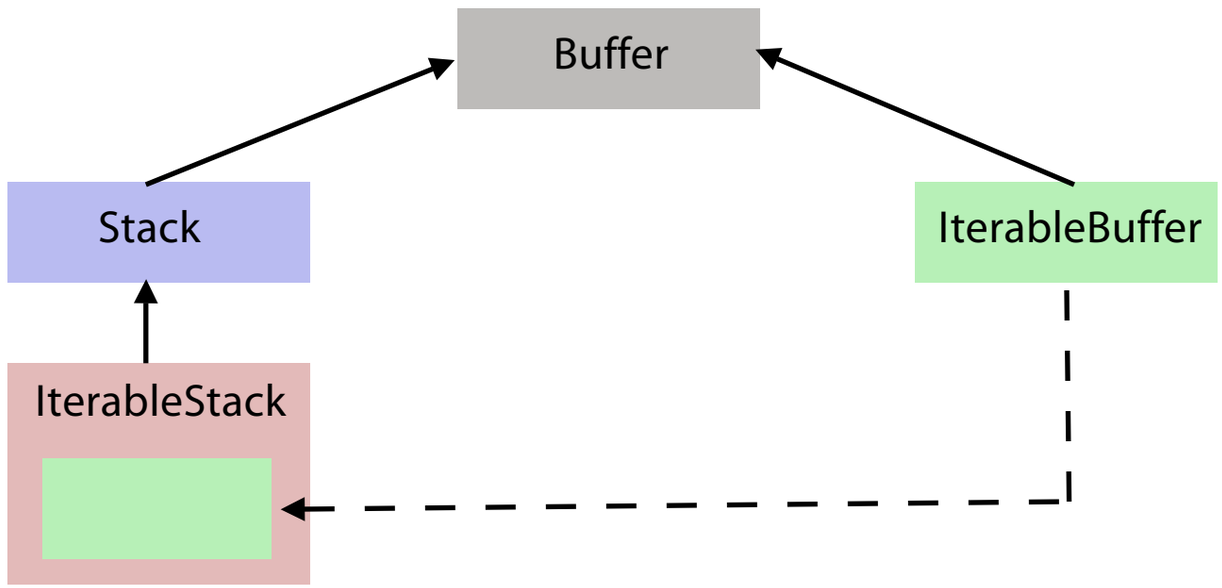
- How to join independent extensions to get a class with combined functionality?
- Solution in Scala: symmetric mixin composition

```
class IterableStack[T] extends Stack[T] with IterableBuffer[T];
```

Superclass

Mixin

- The mixin composition mechanism of Scala allows programmers to reuse the delta of a class definition in the definition of a new class.



Mixin Rules

- Mixin composition in Scala is *symmetric*:
  - *S with A with B = S with B with A*
- S serves as actual superclass of A and B, replacing their declared superclass.
- S must be a subtype of A and B's declared superclass.
- Here's how mixin members are determined:
  - A concrete definition in S, A, or B implements any abstract definitions of the same name in the other classes.
  - Concrete definitions in either A or B override concrete definitions in S.
  - Two concrete definitions of the same name in A and B constitute a conflict, which needs to be resolved in the inheriting class.

Example: Ambiguities

- Imagine we have another extension of class Buffer:

```
class ComparableBuffer[T] extends Buffer[T] {
  def forall(p: T => Boolean): Boolean = ...
  def sameElements(b: IterableBuffer[T]): Boolean =
    forall(elem => ...)
}
```

- ...and we want to integrate it into our stack abstraction:

```
class MyStack[T] extends Stack[T]
                 with IterableBuffer[T]
                 with ComparableBuffer[T];    // ambiguous: forall
```

- Ambiguities have to be resolved by hand:

```
class MyStack[T] extends Stack[T]
                 with IterableBuffer[T]
                 with ComparableBuffer[T] {
  override def forall(p: T => Boolean) =
    super[IterableBuffer].forall(p);
}
```

Diamond Inheritance ⇒ Traits

- To avoid the duplication of state, Scala does not allow a class to be mixed into another class multiple times
- Traits = abstract classes that do not encapsulate state (no variables, no constructor parameters)
- Inheriting twice from a trait is legal
- Example:

```
trait Iterable[T] {
  def elements: Iterator[T];
  def forall(p: T => Boolean): Boolean = {
    val it = elements; var res = true;
    while (res && it.hasNext) { res = p(it.next); }
    res
  }
}

class IterableBuffer[T] extends Buffer[T]
                        with Iterable[T];
```

Traits in the Scala Collection Classes

- Role of traits:
  - Define the basic interfaces (like Map, Set, Buffer, etc.):
    - Most methods are concrete; their implementation relies on a minimal set of abstract methods
    - Concrete classes implement the abstract methods; concrete trait methods are only overridden to improve performance
  - Define optional functionality (like synchronization, checkpointing, etc.)
    - Some/all concrete methods of the basic interface are overridden with alternative implementations
- Benefits:
  - The programmer constructs data types from small building blocks depending on his individual requirements
  - Improved code reuse through more generic class abstractions

Overriding Abstract Members

- Consider a mixin for synchronization:

```
trait Iterator[T] {
  def hasNext: Boolean;
  def next: T;
  def foreach(f: T => unit): unit = while (hasNext) { f(next) };
}

trait SynchronizedIterator[T] extends Iterator[T] {
  abstract override def hasNext: Boolean = synchronized { super.hasNext }
  abstract override def next: T = synchronized { super.next }
}
```

- Problem: super calls in SynchronizedIterator reference abstract members of Iterator.
- Hence an instance creation such as

  `new Iterator[String] with SynchronizedIterator[String]`

  should be illegal.
- This is flagged by the *abstract override* modifier combination.

Dynamic vs. Static Composition

```
class ListIterator[T](xs: List[T]) extends Iterator[T] {
  var ys = xs;
  def hasNext: Boolean = !ys.isEmpty;
  def next: T = { val res = ys.head; ys = ys.tail; res }
}
```

```
class IteratorProxy[T](ip: Iterator[T]) extends Iterator[T] {
  def hasNext: Boolean = ip.hasNext;
  def next: T = ip.next;
}
```

Dynamic Composition

```
new IteratorProxy(new ListIterator(xs))
    with SynchronizedIterator;
```

Static Composition

```
new ListIterator(xs)
    with SynchronizedIterator;
```