Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

November 12, 2019

Traits

Traits

Traits provide a way to factor out common behavior among multiple classes and “mix” it in where appropriate

Trait Definitions

Syntactically, a trait definition looks like an abstract class definition, but with the keyword “trait”:

```
trait Echo {
  def echo(message: String) =
    message
}
```

Trait Definitions

- Traits can declare fields and full method definitions
- They must not include constructors

```
trait Echo {
  val language = "Portuguese"
  def echo(message: String) =
    message
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
class Parrot extends Echo {
  def fly() = {
    // forget to fly and talk instead
    echo("poly wants a cracker")
  }
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
class Parrot extends Bird with Echo {
  def fly() = {
    // forget to fly and talk instead
    echo("poly wants a cracker")
  }
}
```

Using Traits

- Classes "mix in" traits using either the `extends` or `with` keywords

```
trait Smart {
  def somethingClever() =
    "better a witty fool than a foolish wit"
}
```

Using Traits

- Classes can mix in multiple traits via multiple `with`s:

```
class Parrot extends Bird with Echo
with Smart {
  def fly() = {
    // forget to fly and talk instead
    echo(somethingClever())
  }
}
```

Using Traits

Classes can mix in multiple traits via multiple `with`s:

```
trait X
case class Foo()

new Foo() with X
```

*Must use the **new** keyword when creating a new class instance with a **mixin** trait*

Traits with Self-Types

- We can restrict a trait so that it's only valid when mixed-in with a specific type
  - Useful for declaring extra dependencies

```
trait SmartTalk { this: Echo with Smart =>
  def talk() =
    echo(somethingClever)
}
```

Self-Types vs Inheritance

- What is the difference between *extends* and self-types?

  Whereas *extends* introduces a subtype relationship, self-types only specify a dependency.

- When would you *need* to use a self-type
  (i.e., an example where *extends* wouldn't work)?

  Self-typing allows introduction of a *cyclic* dependency between two types. Cyclic subtyping is not possible.

The Diamond Problem

Diamond Inheritance

- Some languages that support unrestricted multiple inheritance (e.g., C++) suffer from the *diamond problem.*

- Simplest example: class's parents have the same parent.

- The problem: If a single class appears multiple times in a class's ancestry, how do you handle that?

- Example (not valid Scala):

```
class A(val x: Int)
class B extends A(4)
class C extends A(5)
class D extends A, B
```

Scala Multiple Inheritance

- Scala does not allow unrestricted multiple inheritance
- Each type can *extend* exactly one supertype, but can also "mix in" one or more *traits* using the *with* keyword:

```
class A(val x: Int)
trait B
trait C
class D extends A with B with C
```

Traits and Diamonds

- Scala solves the diamond problem by ensuring that each constructor is invoked exactly once.

- The same is true for super-method calls.

- Since traits don't take constructor arguments, subclasses can't invoke them with conflicting parameters.

```
trait A
trait B extends A
trait C extends A
class X extends B with C  // OK!
```

Linearization

- To ensure predictable behavior with mix-ins, Scala defines an algorithm for defining a *total ordering* over all supertypes of a given type.

- Since this ordering can be viewed as a linear path through the *directed acyclic graph (DAG)* that is the type's inheritance relationships, it is called a *linearization*.

Linearization Rules

- For each type:
  - Traverse traits right to left
    - Process any traits in its trait-supertypes
- https://www.artima.com/pins1ed/traits.html#i-1280910181-1
- https://scala-lang.org/files/archive/spec/2.13/05-classes-and-objects.html#class-linearization

Why Linearization?

- Predictable super-constructor ordering
- Predictable super-method ordering (for stackable methods in mix-ins)
- Well-defined implementation

Aside: Python introduced its C3-linearization algorithm in version 2.3 in 2003. Before that, multiple inheritance in Python was a complete mess.