Traits

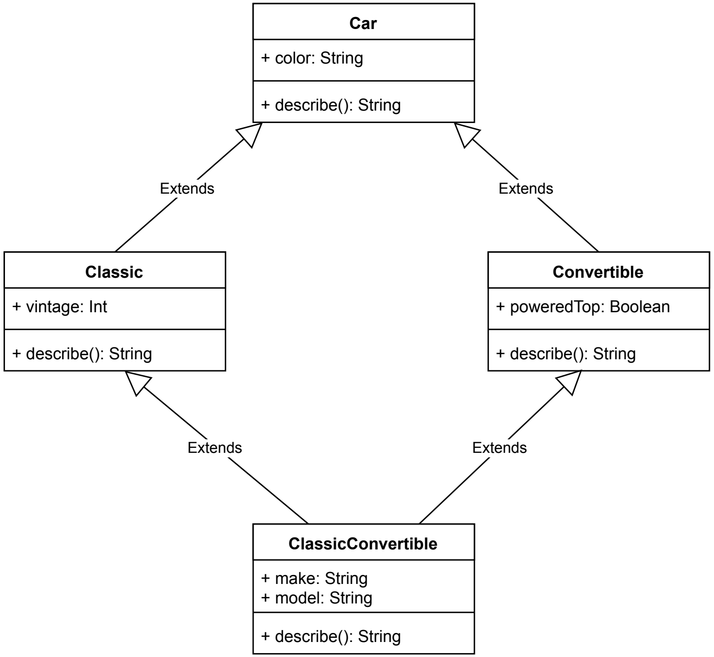
Multiple Inheritance Without the Problems

Agenda

1. Traits Compared To Interfaces
2. Creating a Trait
3. Multiple Traits
4. Linearization
5. Stacking Traits
6. Traits vs Classes
7. Trait Initialization
8. Traits with Type Parameters
9. Selfless Traits

Multiple Inheritance

- The Diamond Inheritance Problem - how does `describe()` visit all instances?



Traits Compared to Interfaces

- Java (and others) sidestep this problem using interfaces
- A class has a single superclass, and multiple interfaces
- Interfaces cannot have state or behavior, only abstract method definitions
- Therefore you cannot accidentally skip behavior, but you also are more limited in where behavior can be inherited from
- In Scala, `traits` are like interfaces (indeed pure abstract traits **are** Java interfaces)
- But they have been extended to include state and behavior
- A class still has a single super-class, but may have multiple traits mixed in as well
- The diamond inheritance problem is tackled in a new, clever way

Creating a Trait

```
trait Car {
  def color: String
  def describe: String = s"$color car"
}
```

- Like a `class` definition but using `trait` keyword instead
- Cannot take constructor parameters
- But can have abstract vals and defs
- Can also have real behavior and state (e.g. describe could be a `def` or `val`)
- Like an abstract class, you cannot make a new instance unless you supply a body

```
val mustang = new Car {
  val color = "red"
}  // Car{val color: String} = $anon$1@5baf4194

mustang.describe  // red car
```

Using a Trait in a Class

```
class ActualCar(val color: String, val name: String) extends Car

val modelT = new ActualCar("black", "Model T")

modelT.describe  // black car
```

- You can extend a `trait` like a superclass, for syntactic convenience
- In fact, all traits have a single superclass as well, by default `AnyRef`
- When you use `extends` for a `trait` you are really extending the trait superclass and mixing in the trait. E.g. the above is really:

```
class ActualCar(val color: String, val name: String) extends AnyRef with Car
```

- Only a `trait` can go after the `with` keyword, not a class

Polymorphism and Rich Interfaces

- Can still use a `trait` like an `interface` to give us polymorphism:

```
val car: Car = modelT
car.describe // black car
```

- We care, because we get free stuff - implement a little, get a lot
- E.g. `Function1`

```
class Demo extends Car with Function1[String, String] {
  override def color = "red"
  override def apply(v1: String): String = s"$v1 $color"
}

val demo = new Demo
demo("cherry")  // cherry red

val descriptionLength = demo.andThen(_.length)
descriptionLength("cherry") // 10
```

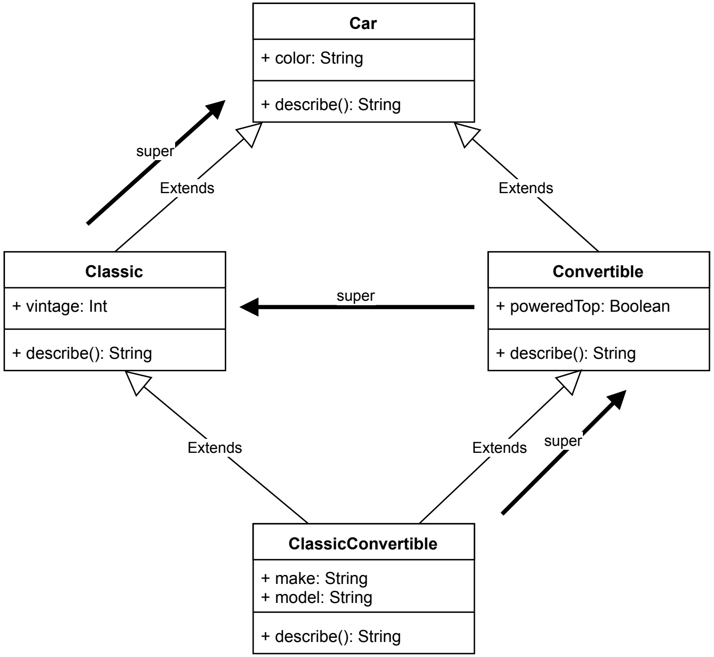
- `andThen` is a method we get for free from `Function1`
- https://www.scala-lang.org/api/2.12.4/scala/collection/Traversable.html

Multiple Traits

```scala
abstract class Car {
  def color: String
  def describe: String = s"$color"
  override def toString = s"$describe car"
}

trait Classic extends Car {
  def vintage: Int
  override def describe: String =
    s"vintage $vintage ${super.describe}"
}

trait Convertible extends Car {
  def poweredTop: Boolean
  override def describe: String = {
    val top = if (poweredTop)
      "powered convertible" else "convertible"
    s"$top ${super.describe}"
  }
}

class ClassicConvertible(
  val color: String, val vintage: Int, val poweredTop: Boolean
) extends Car with Classic with Convertible

val mustang = new ClassicConvertible("red", 1965, false)
// mustang: ClassicConvertible = convertible vintage 1965 red car
```

How'd it do that?

- The `super` is not decided until the `trait` is mixed in to a concrete class
- This is called *linearization*



Stacking Traits

```
abstract class Car {
  def color: String
  def describe: String = s"$color"
  override def toString = s"$describe car"
}

trait Classic extends Car {
  def vintage: Int
  override def describe: String =
    s"vintage $vintage ${super.describe}"
}

trait Convertible extends Car {
  override def describe: String =
    s"convertible ${super.describe}"
}

trait PoweredConvertible extends Convertible {
  override def describe: String =
    s"powered ${super.describe}"
}

trait HardtopConvertible extends Convertible {
  override def describe: String =
    s"hard-top ${super.describe}"
}
```

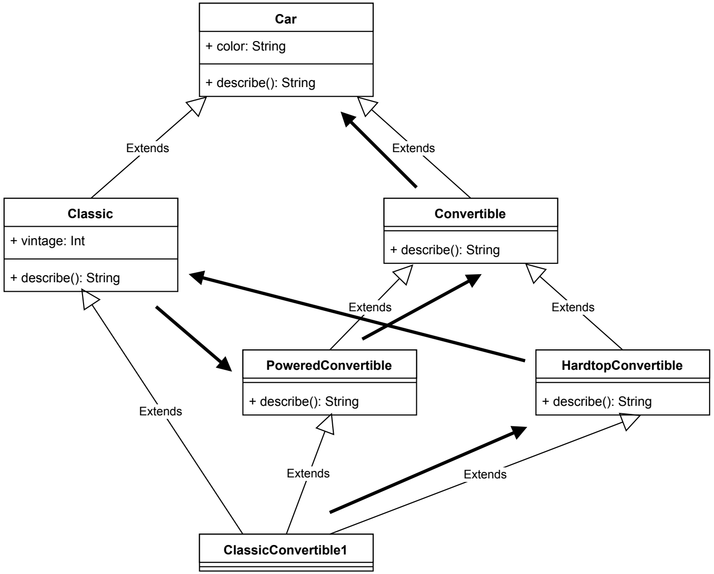
Stacking Traits - Quiz

- What do the following `toStrings` output?

```scala
class ClassicConvertible1(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with Classic with HardtopConvertible

new ClassicConvertible1("red", 1965)


class ClassicConvertible2(val color: String, val vintage: Int)
  extends Car with Classic with PoweredConvertible with HardtopConvertible

new ClassicConvertible2("red", 1965)


class ClassicConvertible3(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with HardtopConvertible with Classic

new ClassicConvertible3("red", 1965)
```

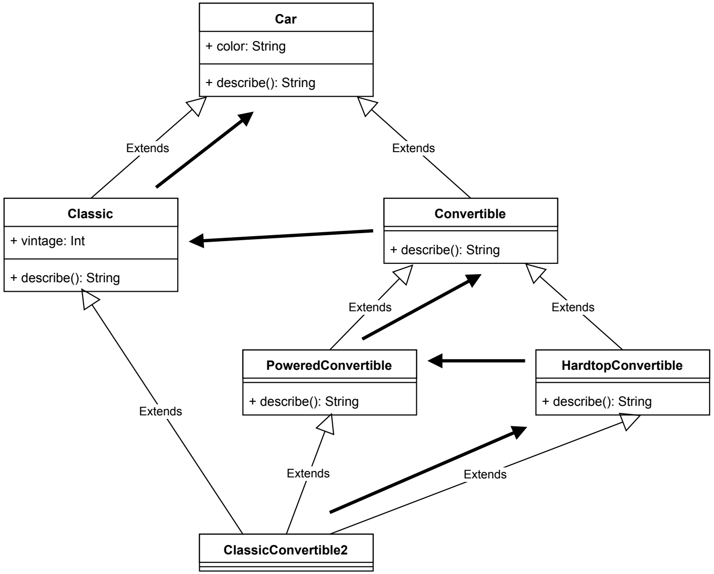
Stacking Traits - 1

```
class ClassicConvertible1(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with Classic with HardtopConvertible

new ClassicConvertible1("red", 1965)
// hard-top vintage 1965 powered convertible red car
```



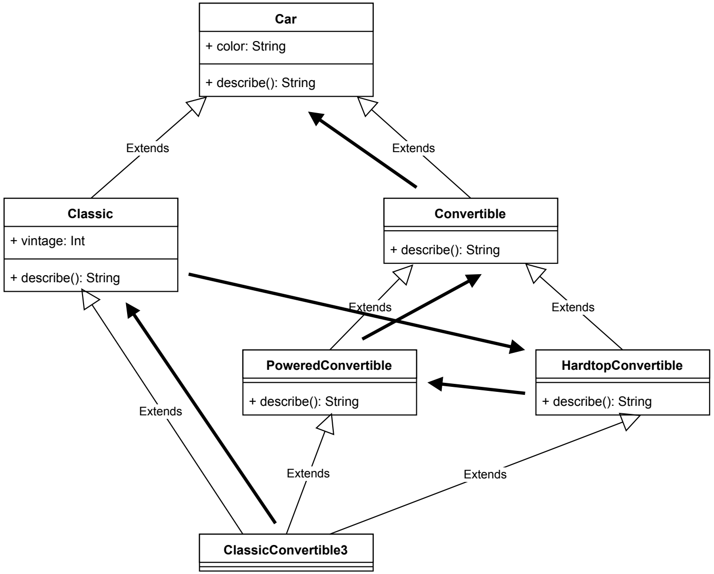
Stacking Traits - 2

```
class ClassicConvertible2(val color: String, val vintage: Int)
  extends Car with Classic with PoweredConvertible with HardtopConvertible

new ClassicConvertible2("red", 1965)
// hard-top powered convertible vintage 1965 red car
```



Stacking Traits - 3

```
class ClassicConvertible3(val color: String, val vintage: Int)
  extends Car with PoweredConvertible with HardtopConvertible with Classic

new ClassicConvertible3("red", 1965)
// vintage 1965 hard-top powered convertible red car
```



Construction Composition

- You can also include traits as you are creating a new instance of a class (just in time)
- This has the effect of introducing a new anonymous sub-class and creating one instance of it immediately, which is given back to you

```
class ClassicCar(val color: String, val vintage: Int) extends Car with Classic

val ccc =
  new ClassicCar("red", 1965) with PoweredConvertible with HardtopConvertible

ccc.describe
// res3: String = hard-top powered convertible vintage 1965 red
```

Traits vs Classes

- Classes (including abstract classes) can have constructor parameters, traits cannot
- This also means traits cannot have implicit parameters or context bounds
- This may change in the future

```
class CoordsC(val x: Double, val y: Double) {
  override def toString: String = s"($x, $y)"
  val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}

val c1 = new CoordsC(3.0, 4.0)  // CoordsC = (3.0, 4.0)
c1.distToOrigin                 // Double = 5.0
```

```
// will not compile, traits can't have constructor params
//trait CoordsT(x: Double, y: Double)
// can use abstract vals instead
trait CoordsT {
  val x: Double
  val y: Double
  override def toString: String = s"($x, $y)"
  val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}
```

Trait Initialization

```
case class Coords(x: Double, y: Double) extends CoordsT

val c2 = Coords(3.0, 4.0)   // Coords = (3.0, 4.0)
c2.distToOrigin             // Double = 5.0
```

- So far so good, but

```
val c3 = new CoordsT {
  val x: Double = 3.0
  val y: Double = 4.0
}  // CoordsT = (3.0, 4.0)

c3.distToOrigin  // Double = 0.0
```

- Huh!?
- `x` and `y` are not set to values until **after** `distToOrigin` has been calculated in the second code snippet

Trait Initialization

- Fixing the problem: Option 1 - early initializers

```
val c4 = new {
  val x: Double = 3.0
  val y: Double = 4.0
} with CoordsT  // CoordsT = (3.0, 4.0)
c4.distToOrigin  // Double = 5.0
```

- Option 2 - use `lazy val` in the trait (recommended)

```
trait CoordsT {
  val x: Double
  val y: Double
  override def toString: String = s"($x, $y)"
  lazy val distToOrigin: Double = math.sqrt((x * x) + (y * y))
}

val c3 = new CoordsT {
  val x: Double = 3.0
  val y: Double = 4.0
} // CoordsT = (3.0, 4.0)
c3.distToOrigin // Double = 5.0
```

- When defining a `trait` make **any** `val` computed from others **`lazy`**

abstract override

- You can override a method in a trait that may be abstract in the superclass, using `abstract override`
- Some other trait must supply a non-abstract implementation in a concrete definition

```
abstract class Vehicle {
  def describe: String  // abstract describe
  override def toString = s"$describe"
}

trait Classic extends Vehicle {
  def vintage: Int
  abstract override def describe: String =
    s"vintage $vintage ${super.describe}"
}

trait Convertible extends Vehicle {
  def poweredTop: Boolean
  abstract override def describe: String = {
    val top = if (poweredTop)
      "powered convertible" else "convertible"
    s"$top ${super.describe}"
  }
}
```

Implementing the Abstract

```
trait Car extends Vehicle {
  def color: String
  def describe: String = s"$color car"  // the actual implementation
}

class ClassicConvertible(
  val color: String, val vintage: Int, val poweredTop: Boolean
) extends Car with Classic with Convertible

val mustang = new ClassicConvertible("red", 1965, false)
// mustang: ClassicConvertible = convertible vintage 1965 red car
```

- Scala will tell you when you get it wrong:

```
Error:(17, 21) method describe in class Vehicle is accessed from super. It may not
be abstract unless it is overridden by a member declared `abstract' and `override'
    s"$top ${super.describe}"
                   ^
```

- There's no magic, someone has to fill in the implementation eventually

Traits with Type Parameters

- Traits can have type parameters:

```
trait CompareAge[T] {
  def older(item: T): T
}

def getOlder[T <: CompareAge[T]](item1: T, item2: T): T = {
  item1 older item2
}
```

```
case class VintageCar(make: String, model: String, year: Int)
  extends CompareAge[VintageCar] {

  def older(other: VintageCar): VintageCar =
    if (this.year < other.year) this else other
}

getOlder(
  VintageCar("Ford", "Mustang", 1965),
  VintageCar("Ford", "Model T", 1922))
// VintageCar(Ford,Model T,1922)
```

Another CompareAge class

```scala
case class Person(name: String, age: Int) extends CompareAge[Person] {
  override def older(other: Person) =
    if (other.age > this.age) other else this
}

getOlder(Person("Fred", 25), Person("Jill", 28))
/// Person(Jill,28)
```

- This is used in the Scala core libraries, e.g. `Ordering`

```scala
val people = List(Person("Fred", 25), Person("Jill", 28), Person("Sally", 22))

people.sorted // Error: No implicit Ordering defined for Person

implicit object PersonOrdering extends Ordering[Person] {
  override def compare(x: Person, y: Person) = x.age - y.age
}

people.sorted
// List(Person(Sally,22), Person(Fred,25), Person(Jill,28))
```

- A `trait` with a single type parameter is often referred to as a *type class*. A widely used pattern in Scala.

Selfless Traits

- Choose trait mixin or import

```scala
trait Logging {
  def error(msg: String): Unit = println(s"Error: $msg")
  def info(msg: String): Unit = println(s"Info: $msg")
}
object Logging extends Logging

class Process1 extends Logging {
  def doIt(): Unit = {
    info("Checking the cell structure")
    error("It's all gone pear shaped")
  }
}
val p1 = new Process1
p1.doIt()

class Process2 {
  import Logging._

  def doIt(): Unit = {
    info("Checking the cell structure")
    error("It's all gone pear shaped")
  }
}
val p2 = new Process2
p2.doIt()
```

Exercises for Module 9

- Find the `Module09` class and follow the instructions to make the tests pass