Composition and Inheritance

Classes, Abstract Classes, extends, super, Overriding rules, final

Agenda

1. Classes and Abstract Classes
2. Uniform Access, `val`, `lazy val` and `def`
3. Inheriting and `extends`
4. Invoking super-class methods and constructors
5. The `override` key word
6. `final` members and classes
7. `case` classes (mention)
8. Domain models

Classes and Abstract Classes

- A class definition can have new instances created for it

```
class Person(name: String, age: Int) {
  def isAdult: Boolean = age >= 21
}

val p1 = new Person("Dave", 18)  // Person@b19efe7
val p2 = new Person("Jill", 25)  // Person@8bb2a08
p1.isAdult // false
p2.isAdult // true
```

- Because it is not marked `abstract`, you are able to create a `new` instance
- Also because it is not marked `abstract`, all fields and methods must have definitions
- When you call `new` in Scala, you **always** get a new instance

```
"hello".eq("hello")                          // true
new String("hello").eq(new String("hello"))  // false
```

- `eq` is instance equality in Scala, while == always calls `.equals`

Abstract Classes

- By contrast, you cannot call `new` on a class marked `abstract`

```
abstract class Car(make: String, model: String, year: Int) {
  def isVintage: Boolean = LocalDate.now.getYear - year > 20
}

val mustang = new Car("Ford", "Mustang", 1965)
// Error: class Car is abstract; cannot be instantiated
// however
val mustang = new Car("Ford", "Mustang", 1965) {} // mustang: Car = $anon$1@7...
```

- When you include an empty body, a new anonymous concrete class is created
- `abstract` classes can also have field and method definitions ommitted:

```
abstract class Car(make: String, model: String, year: Int) {
  def isVintage: Boolean
}
```

Anonymous Classes and Overrides

```
abstract class Car(make: String, model: String, year: Int) {
  def isVintage: Boolean
}

val mustang = new Car("Ford", "Mustang", 1965) {
  def isVintage = LocalDate.now.getYear - year > 20
} // Error: not found: value year
```

- The `year` field referenced in the anonymous class is `private[this]`
- We can make it parametric to get around that:

```
abstract class Car(
  val make: String,
  val model: String,
  val year: Int
) {
  def isVintage: Boolean
}

val mustang = new Car("Ford", "Mustang", 1965) {
  def isVintage = LocalDate.now.getYear - year > 20
}
```

Uniform Access

- In this example, given that `year` is constant, `isVintage` is likely to be constant too

```
abstract class Car(
  val make: String,
  val model: String,
  val year: Int
) {
  val isVintage: Boolean
}

val mustang = new Car("Ford", "Mustang", 1965) {
  val isVintage = LocalDate.now.getYear - year > 20
}
```

- A `val` may override a `def`, but not the other way around
- What happens as the date changes?
- May also use

```
lazy val isVintage = LocalDate.now.getYear - year > 20
```

val, def, lazy val

```
class Demo {
  val a: Int = {
    println("evaluating a")
    10
  }
  def b: Int = {
    println("evaluating b")
    20
  }
  lazy val c: Int = {
    println("evaluating c")
    30
  }
}

val demo = new Demo  // "evaluating a"
demo.a               // res0: Int = 10
demo.b               // "evaluating b" res1: Int = 20
demo.b               // "evaluating b" res2: Int = 20
demo.c               // "evaluating c" res3: Int = 30
demo.c               // res3: Int = 30
```

- `lazy val` calculates if/when first used, them memoizes

Inheriting and Extends

- Classes extend other classes using the `extends` keyword:

```
abstract class Food {
  def name: String
}

abstract class Fruit extends Food

class Orange(val name: String) extends Fruit

val jaffa = new Orange("Jaffa")
```

- `Fruit` must either be `abstract` or provide `name` definition
- `val name: String` parametric field in `Orange` provides `name` override
- New instances of `Orange` can be made, providing the `name` to the constructor

Invoking Super-class Methods/Constructors

```
abstract class Vehicle(val name: String, val age: Int) {
  override def toString: String =
    s"$name, $age years old"
}

class Car(
  override val name: String,
  val make: String,
  val model: String,
  override val age: Int
) extends Vehicle(name, age) {

  override def toString: String =
    s"a $make $model, named ${super.toString}"
}

val mustang = new Car("Sally", "Ford", "Mustang", 50)
// mustang: Car = a Ford Mustang, named Sally, 50 years old
```

- Must `override` the `val`s from the super-class with the same name
- Constructor parameters are passed on through the `extends`
- `super` calls in methods call into the super-class

An Alternative Way to Define Car

```
abstract class Vehicle(val name: String, val age: Int) {
  override def toString: String =
    s"$name, $age years old"
}

class Car(
  name: String,
  val make: String,
  val model: String,
  age: Int
) extends Vehicle(name, age) {

  override def toString: String =
    s"a $make $model, named ${super.toString}"
}

val mustang = new Car("Sally", "Ford", "Mustang", 50)
// mustang: Car = a Ford Mustang, named Sally, 50 years old
```

- If the `override val` feels wierd, you can just make those field `private[this]`
- They will still be public because of the super-class definition
- But you can't make them `vals` in `Car` without an `override`

`override` keyword

- If a `val` or `def` defines a field or method with the same parameter types **over** another of the same name, it must be marked with `override`
- If a `val` or `def` defines a field or method that does not override a superclass field or method with the same parameter types, it must **not** be marked `override`
- If a `val` or `def` defines a field or method with the same parameter types implementing a previously `abstract` field or method, it may or may not be marked `override`

`override` keyword

```
abstract class Superclass {
  def blip: String
  val blop: String = "blop"
  def op(x: Int, y: Int): Int
}

class Subclass extends Superclass {
  override def blip: String = "blip"  // override optional
  override val blop: String = "bloop" // must be override *and* val
  override def op(x: Int, y: Int): Int = x + y  // override optional
  def op(x: Double, y: Double): Double = x + y  // does not override anything
}
```

- Have a play with the worksheet to familiarize yourself better with the rules

`final` keyword

- In Scala, == is always aliased to call `.equals`
- It's tempting to try and override it to do something else:

```
class BadClass {
  override def ==(other: Any): Boolean = {
    println(s"Comparing $this to $other")
    false
  }
}
```

- But if we try:

```
// Error: overriding method == in class Object of type (x$1: Any)Boolean;
// method == cannot override final member
//  override def ==(other: Any): Boolean = {
//               ^
```

- Redefining the meaning of == would be a very bad idea, so it is marked `final` in `AnyRef`

`final` keyword

- We can do the same with our own classes

```
class Authority {
  final def theWord: String =
    "This is the final word on the matter!"
}

class Argumentative extends Authority {
  override def theWord: String =
    "No, it's not!"
}

// Error: overriding method theWord in class Authority of type => String;
// method theWord cannot override final member
//  override def theWord: String =
//               ^
```

`final` classes

- A whole class can be marked final as well, e.g. Java's `String` class:

```
class BadString extends String

// Error: illegal inheritance from final class String
// class BadString extends String
//                         ^
```

- And again, we can do this ourselves:

```
final class Infinity

class Beyond extends Infinity

// Error: illegal inheritance from final class Infinity
// class Beyond extends Infinity
//                      ^
```

- Sorry Buzz Lightyear…

`case` Classes (mention)

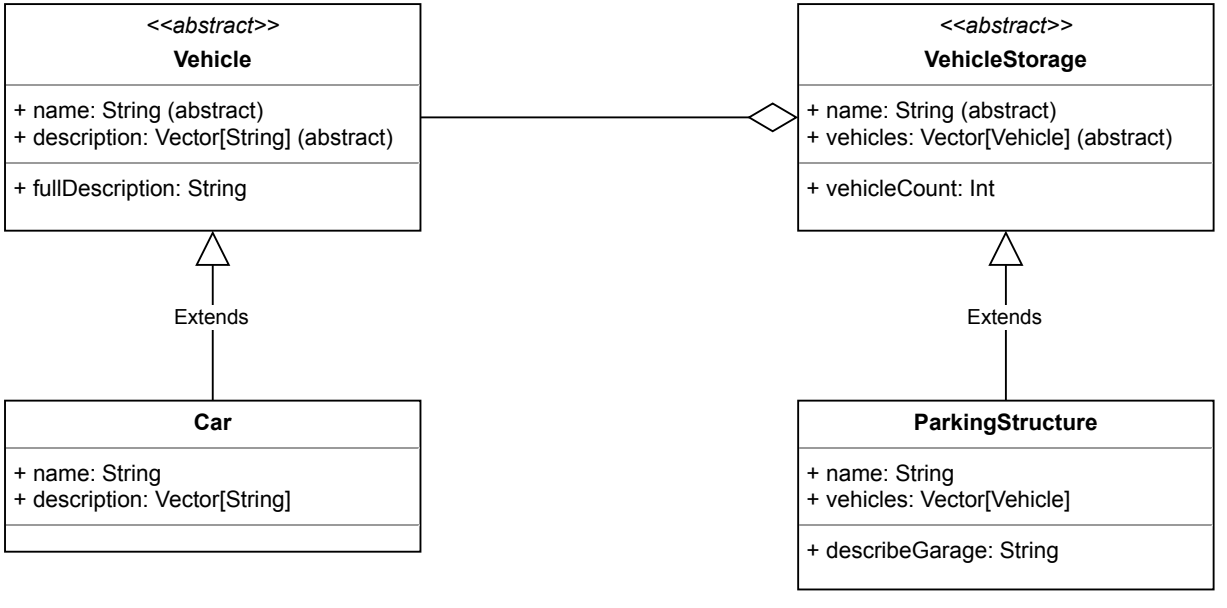
- We'll dig into case classes later in the course, but giving a quick look now:

```
case class Car(make: String, model: String, year: Int) {
  lazy val isVintage: Boolean =
    LocalDate.now.getYear - year > 20
}

val mustang = Car("Ford", "Mustang", 1965) // Car(Ford,Mustang,1965)

mustang.make        // Ford
mustang.model       // Mustang
mustang.year        // 1965
mustang.isVintage   // true

mustang == Car("Ford", "Mustang", 1965)  // true
mustang == Car("Ford", "Mustang", 1964)  // false
```

- With the `case` class you get
  - Parametric immutable fields by default (no `val` needed)
  - A nice `toString` method
  - working `equals` and `hashCode`
  - Factory (`apply`) method (no `new` necessary)
  - More (to be seen later)

Domain Models

- These are pure data abstraction modelling class definitions for a domain
- In Scala they can, and often do, have multiple classes in the same file
- Idiomatically they contain only value state and "pure" behavior directly related to the abstract model
- E.g. let's make a Domain Model for Cars and Parking Garages:



Cars and Vehicles

```
abstract class Vehicle {
  def name: String
  def description: Vector[String]
  override def toString: String = s"Vehicle($name)"

  def fullDescription: String = {
    (name +: description).mkString("\n")
  }
}

case class Car(
  name: String,
  description: Vector[String] = Vector.empty
) extends Vehicle

val mustang = Car("Ford Mustang", Vector(
  "1965 Mustang", "Metallic Blue", "302 ci V8"
))  // Vehicle(Ford Mustang)

val datsun = Car("Datsun 280Z", Vector(
  "1982 Datsun 280Z", "Candy Apple Red", "2.8 Liter I6"
)) // Vehicle(Datsun 280Z)

mustang.fullDescription
// Ford Mustang\n1965 Mustang\nMetallic Blue\n302 ci V8
```

Parking Structure

```
abstract class VehicleStorage {
  def name: String
  def vehicles: Vector[Vehicle]

  def vehicleCount: Int = vehicles.size

  override def toString: String =
    s"$name with $vehicleCount vehicles"
}

case class ParkingStructure(name: String,
  vehicles: Vector[Vehicle]
) extends VehicleStorage {
  def describeGarage: String = {
    val vehicleString = vehicles.mkString(", ")
    s"$name containing $vehicleString"
  }

  override def toString = describeGarage
}

val lot = ParkingStructure(
  "Parking garage",
  Vector(mustang, datsun)
)

lot.vehicleCount // Int = 2
```

Now It's Your Turn: