Composition and Inheritance

Chapter 10

Julia VanLandingham

Otterbein University

February 10, 2021

Two-Dimensional Layout Library

- Elements are rectangles filled with text
- Library provided factory method ”elem"
  - elem(s: String): Element
  - above, beside

Example

```
val column1 = elem("hello") above elem("***")
val column2 = elem("***") above elem("world")
column1 beside column2
```

```
hello ***
*** world
```

Abstract Classes

```
abstract class Element {
   def contents:  Array[String]
}
```

- May have abstract members without implementation
- Cannot be instantiated
- Class has abstract modifier
- Methods do not have abstract modifier

Uniform Access Principle

Client code should not be affected by a decision to implement an attribute as a field or method

Example from Java

`string.length()` not `string.length`

`array.length` not `array.length()`

Parameterless Methods (and when to use them)

- Methods can be defined without any parameter list
  - As opposed to with empty parantheses as *empty-paren methods*
- Typical Conventions
  - Side effects → empty-paren
  - No side effects → parameterless

Example

```
"hello".length   // no () because no side-effect
println()        // better to not drop the ()
```

Extending Classes

```
class ArrayElement(conts:  Array[String]) extends Element{
  def contents:  Array[String] = conts
}
```

- Use the `extends` keyword to *extend* class `Element`
- Scala implicitly assumes your class extends from `scala.AnyRef`
- All members of the superclass are members of the subclass except...
  - Private members of the superclass
  - Members of the superclass with the same name as a member of the subclass

Overriding Methods and Fields

- Overridde a parameterless method with a field
  - values (fields, methods, packages, and singleton objects)
  - types (class and trait names)

Java

```
class CompilesFine {
  private int f = 0;
  public int f() {   return 1;   }
}
```

Scala

```
class WontCompile{
  private var f=0
  def f=1
}
```

Importance of Override

- Helps catch errors like mispelling or incorrect parameters
- Makes for safer system evolution
- Override is...
    - *Required* for members that override a concrete member in a parent class
    - Optional for members that implement abstract members with the same name
    - **Forbidden** for members that do not override or implement some other member in a base class

Parametric Fields

- If you're passing in a parameter just to be copied to a field, something is wrong...
- Use a *parametric field* instead
  - val, var, override, private, public, and protected are options

Before

```
class ArrayElement(conts:  Array[String]) extends Element{
  def contents:  Array[String] = conts
}
```

After

```
class ArrayElement(
  val contents:  Array[String]
) extends Element
```

Pause

Let's pause to put it all together...

```
class Cat{
  val dangerous = false
}

class Tiger(
  override val dangerous:  Boolean,
  private var age:  Int
) extends Cat
```

More Extension

What if our superclass constructor takes a parameter?

```
class LineElement(s:String) extends ArrayElement(Array(s)){
  override def width = s.length
  override def height = 1
}
```

Polymorphism and Dynamic Binding

- We can store any subclass into a variable of the superclass type
  - This is called *subtyping polymorphism*

Example

```
val e1:  Element = new ArrayElement(Array("hello","world"))
val ae:  ArrayElement = new LineElement("hello")
val e2:  Element = ae
```

- Variables and expressions are *dynamically bound*
  - Method implementation is determined at run time based on the actual type of the object not the variable or expression

Final Members (A brief note)

- Use the `final` modifier to prevent any class or member from being overridden or subclassed

Example

```
final class ArrayElement extends Element{ ... }

elem.scala: 18: error: illegal inheritance from final
class ArrayElement
class LineElement extends ArrayElement {
```

Using Composition VS Inheritance

- Generally prefer composition to inheritance
- Ask yourself...
  - Does the inheritance relationship model an *is-a* relationship?
  - Do you expect clients to use the subclass type as a superclass type?
- One of our inheritance relationships looks suspicious...

```
class LineElement(s:  String) extends Element {
  val contents = Array(s)
  override def width = s.length
  override def height = 1
}
```

Implementing above and toString

- We will assume equal heights and widths for simplicity, see section 10.14 for more functionality
- ++ operator concatenates two arrays

```
def above(that:  Element):  Element =
  new ArrayElement(this.contents ++ that.contents)
```

```
override def toString = contents mkString "\n"
```

Implementing `beside`

- First pass...

```
def beside(that:  Element):  Element = {
  val contents = new Array[String](this.contents.length)
  for (i <- 0 until this.contents.length)
   contents(i) = this.contents(i) + that.contents(i)
  new ArrayElement(contents)
}
```

- More functional style

```
new ArrayElement(
  for (
    (line1, line2) <- this.contents zip that.contents
  ) yield line1 + line2
)
```

Factory Method

- Library is simpler for clients to understand
- More opportunities to change library implementation without breaking client code
- Factory method will go inside an Element companion object
  - Import `Element.elem` inside `Element` so we can just call `elem`
  - Move the subclasses to private classes inside the companion object for additional Final results on pages 200-201

Factory Object

```
object Element {
  def elem(contents:  Array[String]):  Element =
    new ArrayElement(contents)

  def elem(line:  String):  Element =
    new LineElement(line)
}
```

Questions?