Imperative Programming 3

Inheritance

Peter Jeavons

Trinity Term 2019

Agenda

- Inheritance: a ***powerful*** tool in OO design
- Inheritance enables polymorphism
- Many uses:
  - separation of “interface” from implementation
  - pulling repeated code into base class
  - extending functionality
- Easy to misuse
  - inheritance vs. composition question

Example from practicals

The editor *Ewoks* stores the text of a file in a `Text` class

```
class Text(init: Int) {
   /* text = buffer[0..gap) ++ buffer[max-len+gap..max) */
   private var buffer = new Array[Char](init)
   private var len = 0
   private var gap = 0
   private def max = buffer.length

   def length = len
   def charAt(pos: Int) = {..}
   def insert(pos: Int, ch: Char) = {..}
   ..
}
```

Dividing text into lines

- If we want to be able to rewrite just one line of the text, then we need some way to extract the current line
- This means we want an *enhanced version* of the `Text` class with more operations (and probably storing more information too, so that we can extract the lines quickly)

Dividing text into lines

Scala allows us to define an extended version of any class, known as a **subclass**

```
class PlaneText extends Text {
    The additional
    instance variables and
    methods go here...
      ...the existing ones from Text are automatically inherited
}
```

Specification for `PlaneText`

STATE: *text : seq char*

lstart : seq int

This records the index in `text` *of the start of each line*

INV: *lstart strictly increasing*

first value = 0

last value = length(text)+1

text[i]='\n' $\Leftrightarrow$ *i+1* $\in$ *lstart*

getRow(pos : int) = r : int

PRE: $0 \leq pos \leq length(text)$

POST: $text=text_0$, $lstart=lstart_0$

$lstart[r] \leq pos < lstart[r+1]$

Implementation for `PlaneText`

- For the concrete state space we could choose:

```
var lstart = new Array[Int](MAXLINES);
var nlines = 0;
// Invariant: 0 ≤ nlines ≤ MAXLINES
// Abs: lstart = lstart[0..nlines)
```

- Or else:

```
var linelen = new Array[Int](MAXLINES);
var nlines = 0;
// Invariant: 0 ≤ nlines ≤ MAXLINES
// Abs: lstart[i] = Sum(linelen[0..i))
```

Question: Which is better? Why?

Implementation for `PlaneText`

Now we consider the methods:

- Some methods of `PlaneText` are simply *inherited* from `Text`, e.g. `charAt`
- Some are *added* and so need to be implemented, e.g. `getRow`
- Some have a *changed* specification, so must be **overridden**, e.g. `insert`
  – it now has to update the line lengths

Implementation for `PlaneText`

- Methods that override methods in the superclass can be identified with the tag `override` (the compiler then checks…)

```
class PlaneText extends Text {
    ...
    def getRow(pos: Int) {...}

    override def insert(pos: Int, ch: Char) {
        Do the same as the original insert method in the superclass Text...
        super.insert(pos,ch)
            ...and then update the linelen values as necessary
    }
}
```

When to use inheritance

- Inheritance can be helpful when B is a kind of A
  - An `Apple` is a kind of `Fruit`
  - An `Ocean` is a kind of `BodyOfWater`
  - A `PlaneText` is a kind of `Text`
  - A `ForwardEulerOdeSolver` is a kind of `OdeSolver`
- The "is-a" relationship should be rigid, so it holds for as long as B exists, but
  - A `Person` is an `Employee` until they are fired
  - A `Footballer` is an `ArsenalFootballer` until they are sold to another club
- Non-rigid relationships should be modelled with roles/attributes, **not** with inheritance

Dynamic Binding

- Consider the following method defined in the class `Text`

```
def insertString(pos: Int, s: String) {
   for (i <- 0 until s.length)
      this.insert(pos+i,s.charAt(i));
}
```

- This method invokes the `insert` method, but it executes a *different version* of it depending on the class of the object on which it is invoked – this is called **dynamic binding**

Dynamic Binding

Static type Dynamic type

```
val t:Text = new PlaneText()
t.insertSring(0,”Hello”)
```

```
def insertString(pos: Int, s: String) {
   for (i <- 0 until s.length)
       this.insert(pos+i,s.charAt(i));
}
```

  - if the `insert(pos,char)` method from `Text` is used that would be bad because the line map won’t update
  - if the `insert(pos,char)` method from `PlaneText` is used that would be better

- Rule: the dynamic type of the object is used to choose which method implementation to invoke

Fragile Base Class

```
val t:Text = new PlaneText()
t.insertSring(0,”Hello”)
```

```
def insertString(pos: Int, s: String) {
   val n = s.length
   makeRoom(n); moveGap(pos)
   s.getChars(0, n, buffer, pos)
   gap += n; len += n
}
```

(This uses a fast `String.getChars` method to copy characters: really efficient!)

But `PlaneText` no longer works!

Bug history

1. Introduce `Text` class with `insertString` method which calls `insertChar` repeatedly
2. Derive `PlaneText` subclass which has line map
3. Application developer writes application which uses `PlaneText` and calls `insertString` (works well)
4. Original developer of `Text` re-implements `insertString` to be more efficient
5. Application breaks (and original developer doesn’t know about it)

Fragile Base Class

- This kind of error is sometimes known as the "fragile base class" problem, because changing an implementation can break the program, if subclasses happen to rely on a particular implementation

- This kind of issue gives rise to another slogan of OOP:

“Inheritance is Dangerous”

Moral

“Inheritance is Dangerous”

- Inheritance is dangerous when the superclass is a concrete class that is subject to change
  - even if the change doesn’t alter its specification or externally observable behaviour
- The underlying issue is that the interface between a class and its subclasses is hard to specify precisely – subclasses break encapsulation

Inheritance versus composition

- We can avoid the fragile base class problem by composing objects together: make method calls explicit
- For example we could make `PlaneText` a container which devolves all its decisions to a specified `Text` object:

```
class PlaneText {
   private val text:Text = new Text();
   def length = text.length
```

- Benefit: the two classes are more loosely coupled
- Benefit: the delegated member `text` can be managed
- Problem: all calls to `PlaneText` have to be forwarded (someone has to write a lot of methods)
- Problem: no polymorphism so difficult to substitute

Overriding and overloading

- Note: in the actual class `Text` there is no `insertString` method: inserting a character or a string are both done by a method called “insert”

```
def insert(pos: Int, c: Char) {...}
def insert(pos: Int, s: String) {
   for (i <- 0 until s.length)
       this.insert(pos+i,s.charAt(i));
}
```

Same name different parameter types

- The runtime system distinguishes methods with the same name based on:
  - the types of the parameters (*overloading*)
  - and the dynamic type of the object (*overriding*)

Overriding and overloading

- For the sake of this discussion it was less confusing to pretend that the class `Text` had an `insertString` method
- Some people say method overloading is dangerous because it becomes less clear which method will actually be run

Question: How does the runtime system decide between an *overriden* method and an *overloaded* method?

An aside: the Any class

- *Every* class in Scala is a subclass of the **`Any`** class
- Hence every object has methods that are inherited from `Any`
- These include:
  - `toString()` which returns a String representing the object
  - `equals(b: Any)` which compares the object with another
- These methods should be over-ridden if the default behaviour is not appropriate

Aside: identity versus equality

Objects can be checked for “sameness”

- By asking if they are the same object (identity or reference equality)
- By asking if they have the same content

```
class Point (val x:Int, val y:Int)
   val a = new Point(0,0)
   val b = a  // b is just an alias for a

   // Identity check
   a eq b // true

   // Content check
   a equals b // true
   a == b     // true (short for ‘equals’)
```

Aside: identity versus equality

To check content equality, override `equals`

```
class Point (val x:Int, val y:Int){
   override def equals(other: Point): Boolean =
     this.x == other.x && this.y == other.y
}

val a = new Point(0,0)
val b = a  // b is just an alias for a
val c = new Point(0,0) // c has same content

// Identity checks
a eq b // true
a eq c // false

// Content checks
a == b // true
a == c // true  <- using equals
```

Need to make this more general

Overriding `equals()`

- The method `equals()` should
  - Compare to any object not just other points

```
override def equals(other: Any): Boolean =
{ .. // return false if ‘other’ is not a Point}
```

  - Be an equivalence relation (reflexive, symmetric, transitive and consistent)
  - Accompany a change to the `hashCode` Scala’s shortcut for object contents
    - Equal content gets same hash, but not vice versa

```
override def hashCode = 29*x + y
```

Summary

- Inheritance is powerful
- Dynamic binding (versus static types)
- Inheritance can be dangerous
  - Fragile base class
- Inheritance versus composition
- Overriding versus overloading
- Identity versus equality

See also *Programming in Scala*: Chapters 10,11 & 30

- Next time: Design patterns