Imperative Programming 2:
Inheritance 2

Hongseok Yang
University of Oxford

Plan

- Yesterday: Basic principles for inheritance (Chap 10).
  - Code reuse by inheriting methods and fields.
  - Subtyping.
  - Overriding and parameterisation.
- Today: Example (Chap 10).
  - We will develop a two-dimensional layout library.

Learning outcome

- To gain an experience of using inheritance for organising classes.
- Can use two programming idioms:
  - Parameterise and instantiate a class via inheritance and overriding.
  - Factory method used for information hiding.

Programming challenge

- Implement a library for two-dimensional layouts.

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

Programming challenge

- Implement a library for two-dimensional layouts.

```
val col1 = elem(Array("71", "--", "10"))
val layout = col1
println(layout)
```

```
71
--
10
```

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

Programming challenge

- Implement a library for two-dimensional layouts.

```
val col1 = elem(Array("71", "--", "10"))
val col2 = elem(" x ") beside elem("9")
val layout = col1 beside col2
println(layout)
```

```
71
-- x 9
10
```

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

In Scala, "`obj m x`" means the same as "`obj.m(x)`".

Programming challenge

- Implement a library for two-dimensional layouts.

```
val col1 = elem(Array("71", "--", "10"))
val col2 = elem(" x ") beside elem("9")
val row2 = elem('-', 8, 1) above elem("1232")
val layout = (col1 beside col2) above row2
println(layout)

 71
 -- x 9
 10
--------
  1232
```

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

In Scala, "`obj m x`" means the same as "`obj.m(x)`".

Programming challenge

- Implement a library for two-dimensional layouts.

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

High-level plan

- Three ways for creating an object of type `Element`.
- We will implement them using three subclasses of `Element`.

```
class Element {
  ...
  def toString ...
  def above ...
  def beside ...
}

object Element {
  ...
  def elem(contents: Array[String]): Element = ...
  def elem(chr: Char, width: Int, height: Int): Element = ...
  def elem(s: String): Element = ...
}
```

Class hierarchy

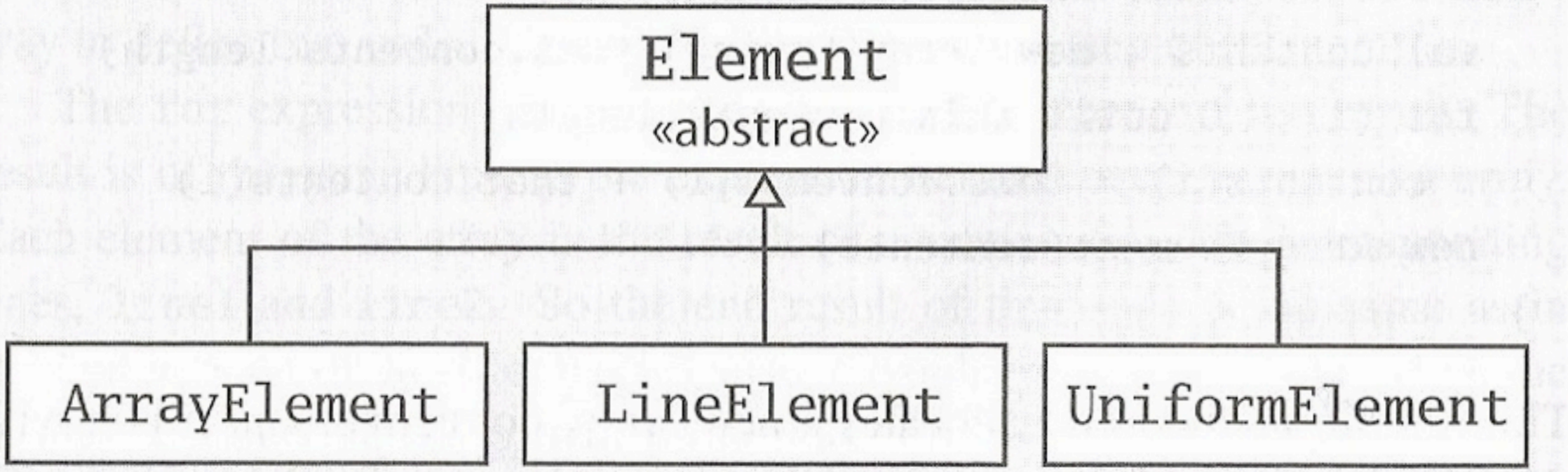


Figure 10.4 · Class hierarchy with revised `LineElement`.

Class hierarchy

Implements `toString`, `above` and `beside`, which are parameterised by a method `contents`.

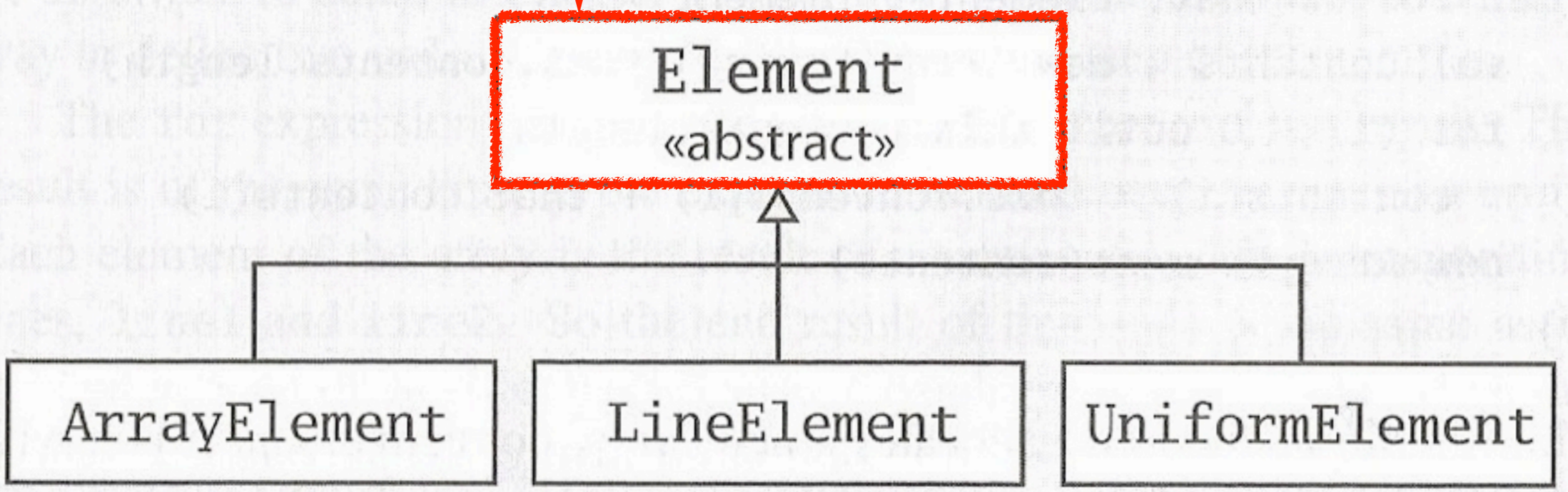
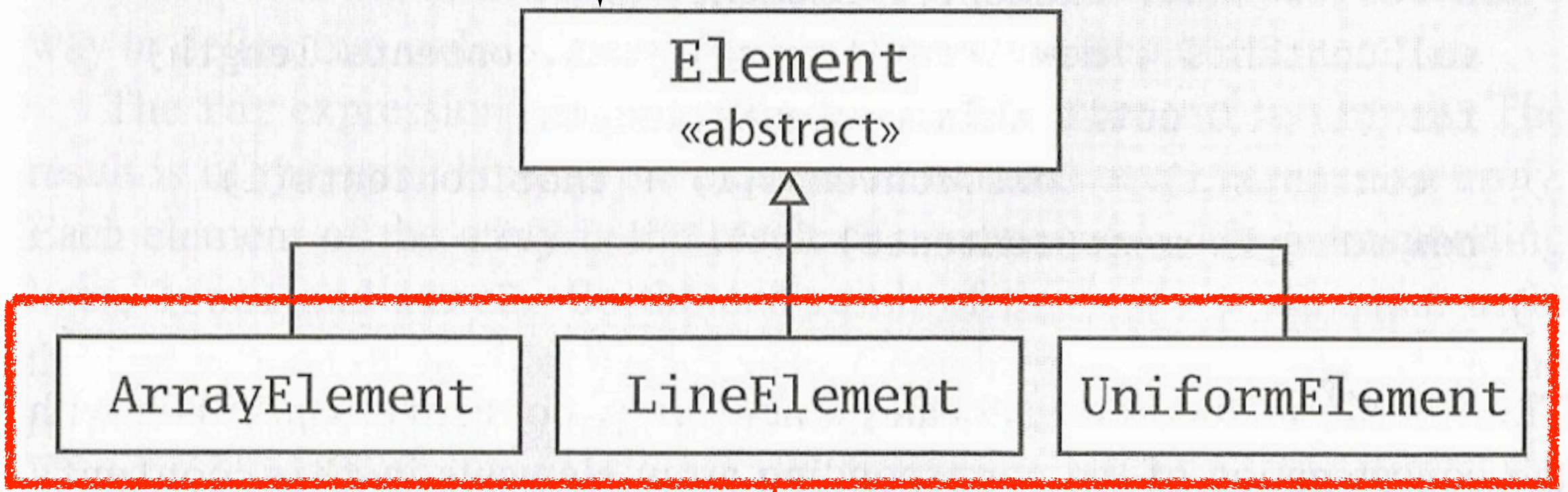


Figure 10.4 · Class hierarchy with revised `LineElement`.

Class hierarchy

Implements `toString`, `above` and `beside`, which are parameterised by a method `contents`.



Override `contents`. They are used to implement three `elem` methods.

Class `Element`

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

  override def toString = contents mkString “\n“
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}
```

- `Element` declares the method `contents`, which returns an array for storing each row of a layout.
- But `contents` is not defined.
- `Element` is an abstract class, which is parameterised by the definition of `contents`.

Overriding parameterless method

- A parameterless method "`def f:C`" in a superclass can be overriden by a field "`val f:C`".
- Subtle difference:

```
val f: Array[Int] = Array(1,2,3)
```

Create an array only once during object creation.

```
def f: Array[Int] = Array(1,2,3)
```

Create an array whenever `f` is accessed.

Overriding in subclasses

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

class ArrayElement(c: Array[String]) extends Element {
  val contents = c
}

class LineElement(...) extends Element ...

class UniformElement(...) extends Element ...
```

In Scala, “`val f:D`” can override “`def f:D`”.

Overriding in subclasses

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

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

class LineElement(...) extends Element ...

class UniformElement(...) extends Element ...
```

“class C(val x: D) ...” means the same as
“class C(tempX: D) ... { val x: D = tempX ...}”

Exercise: Fill in the boxes

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

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

class LineElement(s: String) extends Element {
  val contents =
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents =
}
```

Exercise: Fill in the boxes

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents =
}
```

Exercise: Fill in the boxes

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Exercise: Fill in the boxes

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

  override def toString = contents mkString "\n"
  def above(that: Element): Element = ...
  def beside(that: Element): Element = ...
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

How are "`val contents`" and "`def contents`" different?

above and beside

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

  override def toString = contents mkString “\n”
  def above(that: Element): Element =

  def beside(that: Element): Element =



}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Implementation of above

Assume "this" and "that" have the same width.

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

  override def toString = contents mkString "\n"
  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element): Element =



}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Exercise: Implement `beside`

We assume "`this`" and "`that`" have the same height.

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

  override def toString = contents mkString "\n"
  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element): Element =



}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Exercise: Implement `beside`

We assume "`this`" and "`that`" have the same height.

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

  override def toString = contents mkString "\n"
  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element): Element =
    new ArrayElement(
      for ((line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2)
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Full implementation

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

  override def toString = contents mkString "\n"
  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element): Element =
    new ArrayElement(
      for ((line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2)
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Full implementation

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


  def elem(chr: Char, width: Int, height: Int): Element =


  def elem(line: String): Element =

}
```

```
abstra
  def

  over
  def
    ne
  def
    ne
    ) yield line1 + line2)
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

Full implementation

```
abstra
  def

  over
  def
    ne
  def
    ne
      ) yield line1 + line2)
}

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

class LineElement(s: String) extends Element {
  val contents = Array(s)
}

class UniformElement(ch: Char, w: Int, h: Int) extends Element {
  def contents = Array.fill(h)(ch.toString * w)
}
```

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

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

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

Full implementation

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

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

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

Here we are using subtyping relationships, such as

`ArrayElement <: Element`,

which are induced by inheritance.

Factory method

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

  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)

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

- A method whose main aim is to create an object.
- Decides the kind of an object to create, such as its type.
- This decision depends on method arguments, and is normally hidden from callers.

Unfinished business I

- `ArrayElement`, `LineElement`, `UniformElement` are conceived to implement the `elem` methods.
- How can we hide these implementation details from the callers of the `elem` methods?
- Any suggestion?

Hiding implementation details

```
object Element {


  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)
  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)
  def elem(line: String): Element =
    new LineElement(line)
}
```

Hiding implementation details

1. Make subclasses private to the singleton object.

```
object Element {
  private class ArrayElement(val contents: Array[String])
    extends Element
  private class LineElement(s: String)
    extends Element { val contents = Array(s) }
  private class UniformElement(ch: Char, w: Int, h: Int)
    extends Element { def contents = Array.fill(h)(ch.toString*w) }

  def elem(contents: Array[String]): Element =
    new ArrayElement(contents)
  def elem(chr: Char, width: Int, height: Int): Element =
    new UniformElement(chr, width, height)
  def elem(line: String): Element =
    new LineElement(line)
}
```

Hiding implementation details

1. Make subclasses private to the singleton object.
2. Ensure that objects of these subclasses are created only by the factory methods.

```
object El
  private
    exten
  private
    exten
  private
    exten

  def ele
    new A
  def ele
    new U
  def ele
    new L
}
```

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

  override def toString = contents mkString “\n“
  def above(that: Element): Element =
    new ArrayElement(this.contents ++ that.contents)
  def beside(that: Element): Element =
    new ArrayElement(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )
}
```

Hiding implementation details

1. Make subclasses private to the singleton object.
2. Ensure that objects of these subclasses are created only by the factory methods.

```
object El
  private
    exten
  private
    exten
  private
    exten

  def ele
    new A
  def ele
    new L
  def ele
    new L
}
```

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

  override def toString = contents mkString “\n“
  def above(that: Element): Element =
    Element.elem(this.contents ++ that.contents)
  def beside(that: Element): Element =
    Element.elem(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )
}
```

Hiding implementation details

1. Make subclasses private to the singleton object.
2. Ensure that objects of these subclasses are created only by the factory methods.

```
object El
  private
    exten
  private
    exten
  private
    exten

  def ele
    new A
  def ele
    new L
  def ele
    new L
}
```

```
import Element.elem

abstract class Element {
  def contents: Array[String]

  override def toString = contents mkString “\n“
  def above(that: Element): Element =
    elem(this.contents ++ that.contents)
  def beside(that: Element): Element =
    elem(
      for (
        (line1, line2) <- this.contents zip that.contents
      ) yield line1 + line2
    )
}
```

Unfinished Business 2

- We made same width and height assumptions in our implementation of `above` and `beside`.
- Consequently, we get a wrong output:

```
val col1 = elem(Array("71", "--", "10"))
val col2 = elem(" x ") beside elem("9")
val row2 = elem('-', 8, 1) above elem("1232")
val layout = (col1 beside col2) above row2
println(layout)
```

Actual output

```
71 x 9
--------
1232
```

Intended output

```
 71
 -- x 9
 10
--------
  1232
```

Fixed version of above

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



  override def toString = ...
  def above(that: Element): Element = {


    elem(this.contents ++ that.contents)
  }









  def beside(that: Element): Element = ...
}
```

Fixed version of above

```
abstract class Element {
  def contents: Array[String]
  def width: Int = contents(0).length
  def height: Int = contents.length
  override def toString = ...
  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }
  def widen(w: Int): Element =









  def beside(that: Element): Element = ...
}
```

Fixed version of above

Exercise: Complete the implementation of `widen`.

```
abstract class Element {
  def contents: Array[String]
  def width: Int = contents(0).length
  def height: Int = contents.length
  override def toString = ...
  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }
  def widen(w: Int): Element =









  def beside(that: Element): Element = ...
}
```

Fixed version of above

Exercise: Complete the implementation of `widen`.

```
abstract class Element {
  def contents: Array[String]
  def width: Int = contents(0).length
  def height: Int = contents.length
  override def toString = ...
  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }
  def widen(w: Int): Element =
    if (w <= width) this
    else {
      val left = elem(‘ ‘, (w - width) / 2, height)
      val right = elem(‘ ‘, w - width - left.width, height)
      left beside this beside right
    }
  def beside(that: Element): Element = ...
}
```

Fixed version of `beside`

- Similar implementation to above.
- Look at the textbook.

Exercise: Fix `beside`

```
abstract class Element {
  def contents: Array[String]
  def width: Int = contents(0).length
  def height: Int = contents.length
  override def toString = ...
  def above(that: Element): Element = {
    val this1 = this widen that.width
    val that1 = that widen this.width
    elem(this1.contents ++ that1.contents)
  }
  def widen(w: Int): Element =
    if (w <= width) this
    else {
      val left = elem(‘ ‘, (w - width) / 2, height)
      val right = elem(‘ ‘, w - width - left.width, height)
      left beside this beside right
    }
  def beside(that: Element): Element = ...
  def heighten(h: Int): Element = ...
}
```

Summary

- Inheritance and overriding can be used to implement parameterisation and instantiation.
- Use of factory methods and subtyping for hiding implementation details.
- Read Chap 10.
- Try Scala code in Listings 10.12 - 10.14.