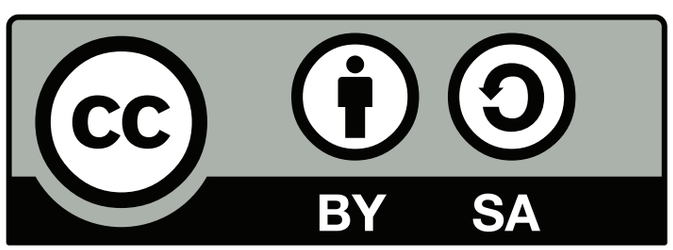
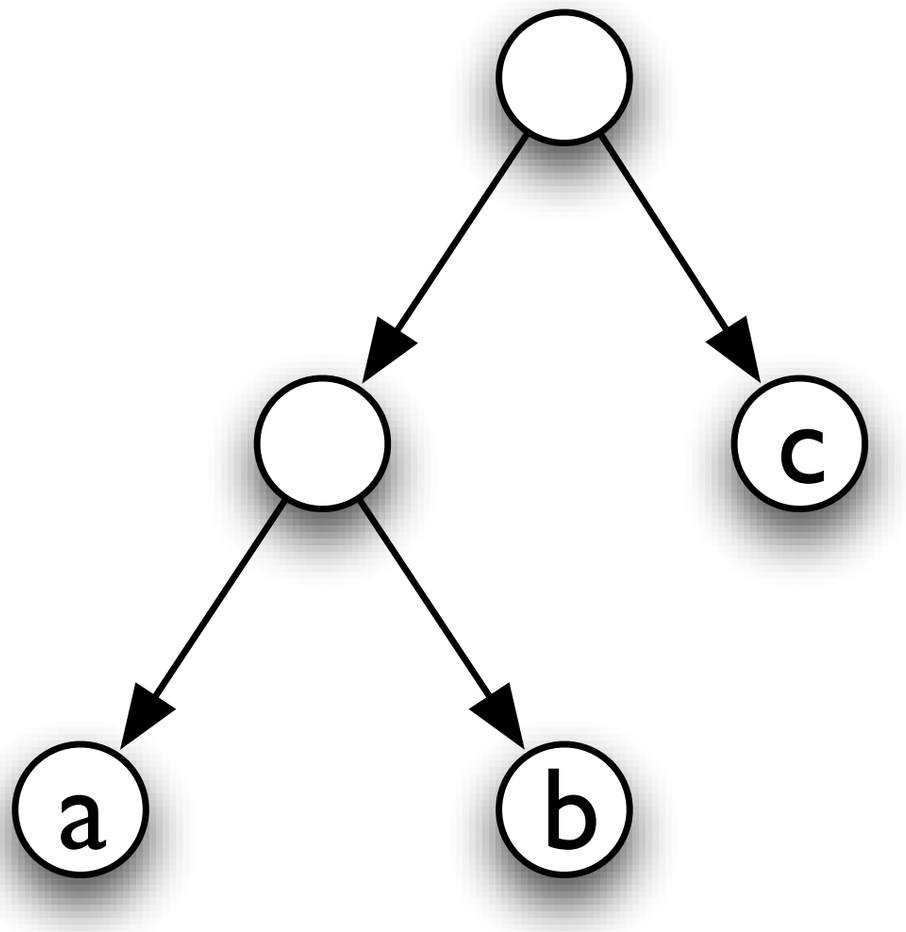
Tecnologia e Applicazioni Internet 2009/10

Lezione 3 - Better views

Matteo Vaccari
http://matteo.vaccari.name/
matteo.vaccari@uninsubria.it

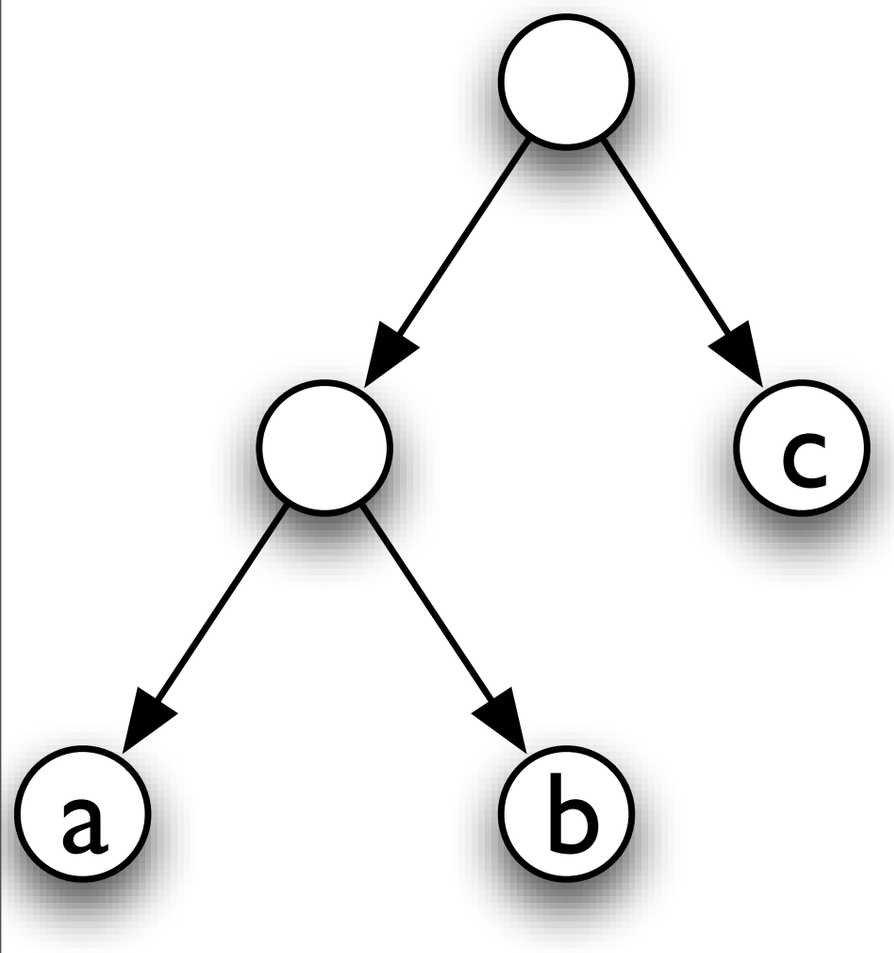





```
abstract class Tree {}

class Leaf extends Tree {
    String label;

    Leaf(String label) {
        this.label = label;
    }
}

class Fork extends Tree {
    Tree left;
    Tree right;

    Fork(Tree left, Tree right) {
        this.left = left;
        this.right = right;
    }
```

```
Tree tree =
    new Fork(
        new Fork(
            new Leaf("a"),
            new Leaf("b")),
        new Leaf("c"));
```

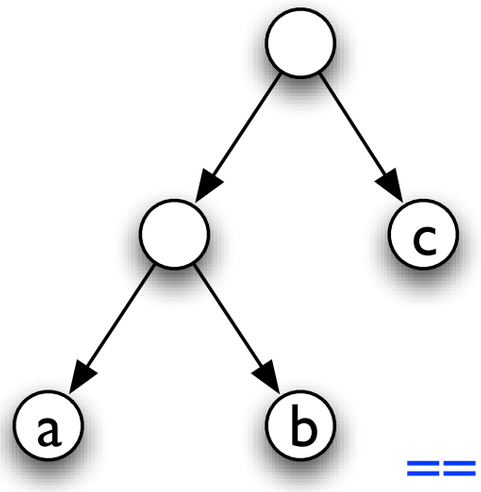
Come si definisce una struttura dati ricorsivamente in Java? Si definisce una classe base astratta che rappresenta il tipo, e si estende per tutti i possibili “casi”. Per esempio un albero binario con etichette sulle foglie si definisce così. Abbiamo il caso “base” che è la foglia, e il caso ricorsivo che è la biforcazione.




```
abstract class Tree {
    abstract int height();
}

class Leaf extends Tree {
    ...
    int height() {
        return 1;
    }
}
```

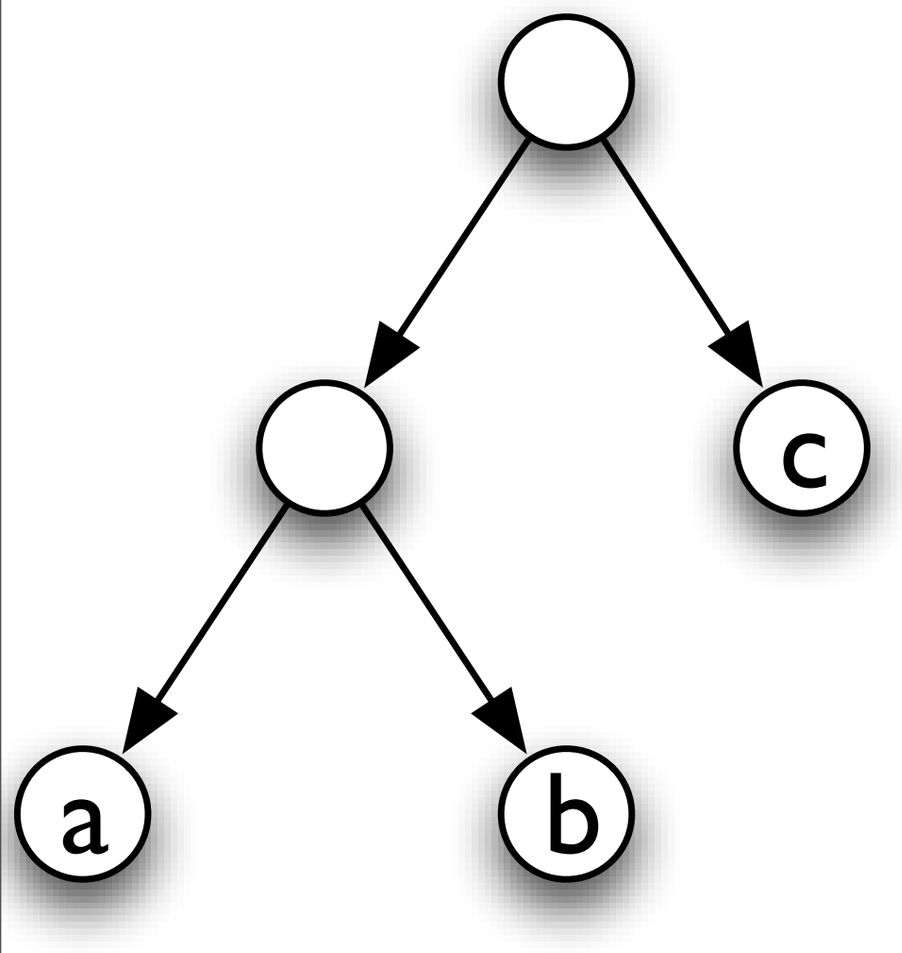
```
class Fork extends Tree {
    ...
    int height() {
        return 1 +
            Math.max(left.height(),
                     right.height());
    }
}
```

Come si definisce una funzione su un tipo definito ricorsivamente? Si definisce in maniera astratta sul tipo astratto. Poi si definisce come valutarla nel caso “base”, e come valutarla nel caso ricorsivo.




```
new Fork(
     new Fork(
          new Leaf("a"), new Leaf("b")),
     new Leaf("c")).height()
==
     1 + Math.max(
     new Fork(new Leaf("a"), new Leaf("b")).height(),
     new Leaf("c").height());
==
     1 + Math.max(
          1 + Math.max(new Leaf("a").height(),
                       new Leaf("b").height()),
          1);
==
     1 + Math.max(1 + Math.max(1, 1), 1);
==
     1 + Math.max(1 + 1, 1);
==
     3
```

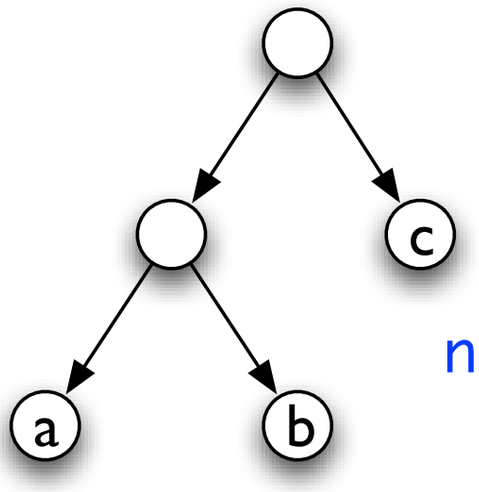
E questo è un esempio di come la funzione height venga computata ricorsivamente.



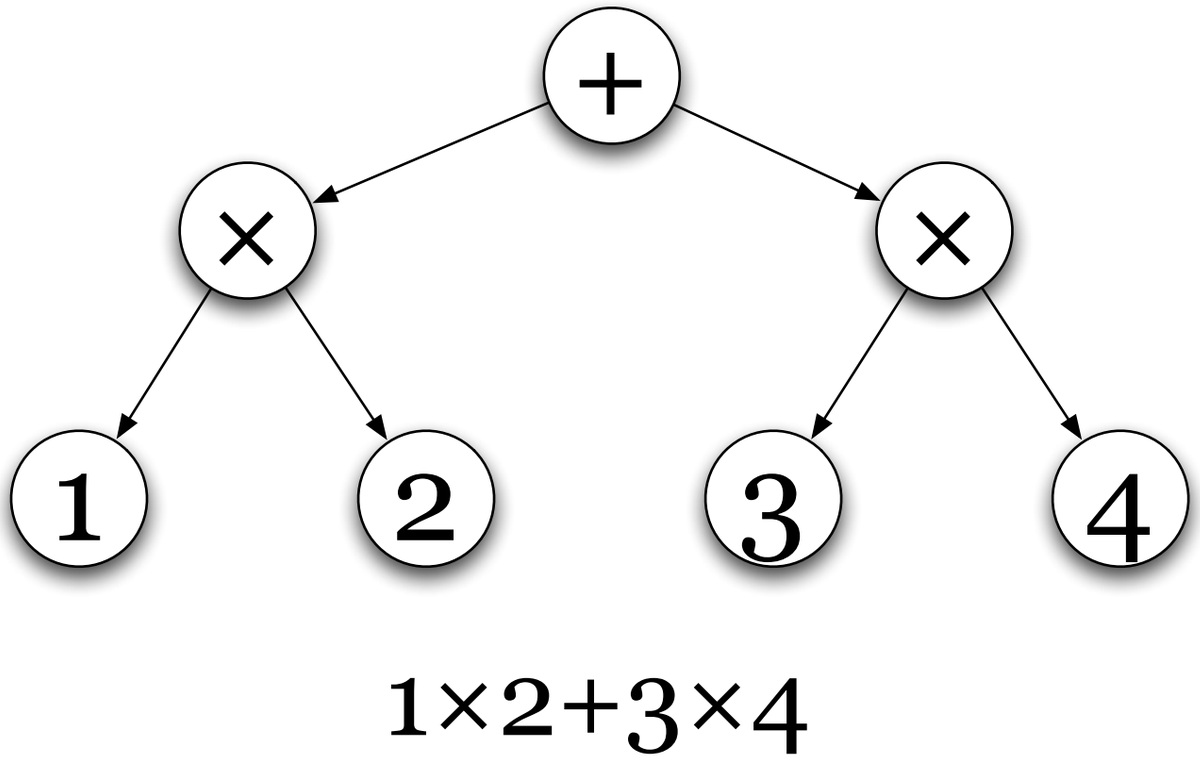
“a, b, c”

```
abstract class Tree {
    ...
    abstract String frontier();
}

class Leaf extends Tree {
    ...
    String frontier() {
        return label;
    }
}

class Fork extends Tree {
    ...
    String frontier() {
        return left.frontier()
            + ", "
            + right.frontier();
    }
}
```

Un’altro esempio di funzione: la “frontiera” dell’albero sono tutte le stringhe sulle foglie, concatenate con una virgola.



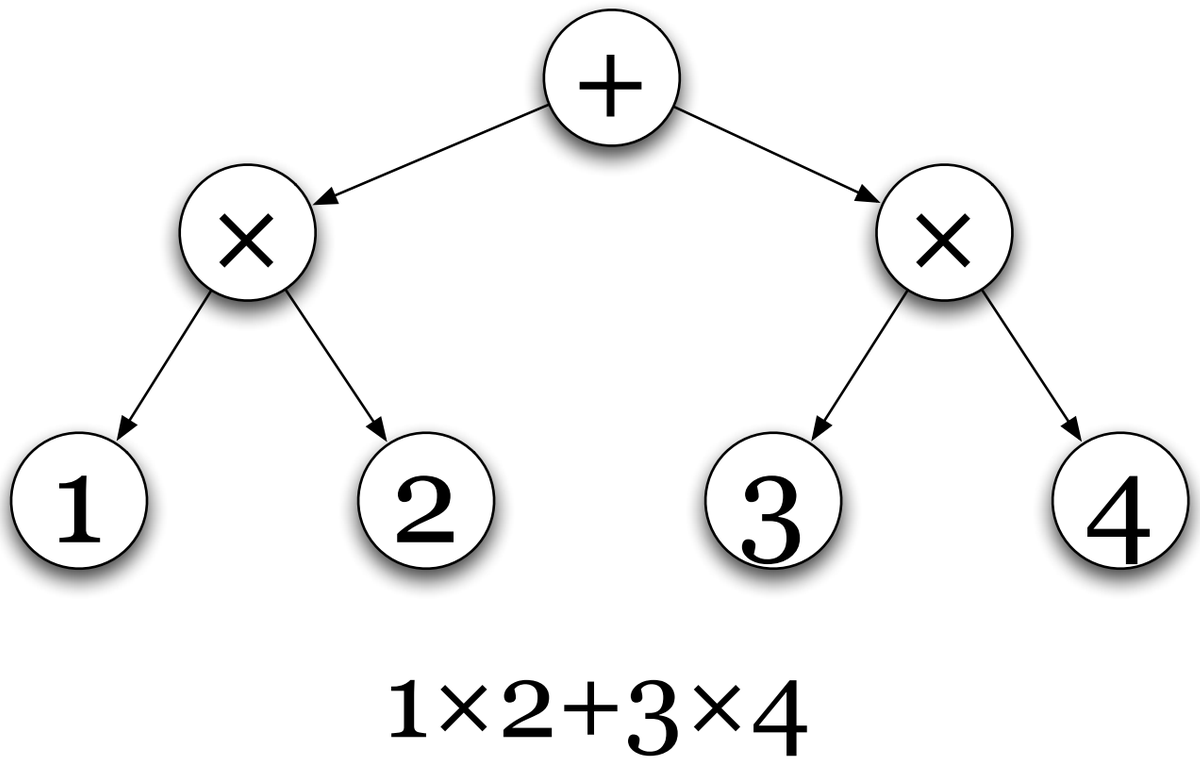

```
new Fork(
    new Fork(
        new Leaf("a"), new Leaf("b")),
    new Leaf("c")).frontier()
==
    new Fork(new Leaf("a"), new Leaf("b")).frontier()
    + ", "
    + new Leaf("c").frontier();
==
    new Leaf("a").frontier() + ", " + new Leaf("b").frontier()
    + ", "
    + "c";
==
    "a" + "," + "b"
    + ", "
    + "c";
==
    "a, b, c"
```




```
abstract class Expression {}

class Constant {
    double value;
    Constant(double value) {
        this.value = value;
    }
}
```

```
class Plus {
    Expression left;
    Expression right;
    Plus(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
}

class Times {
    ...
}
```

Un altro esempio di struttura dati definita ricorsivamente sono le espressioni aritmetiche.

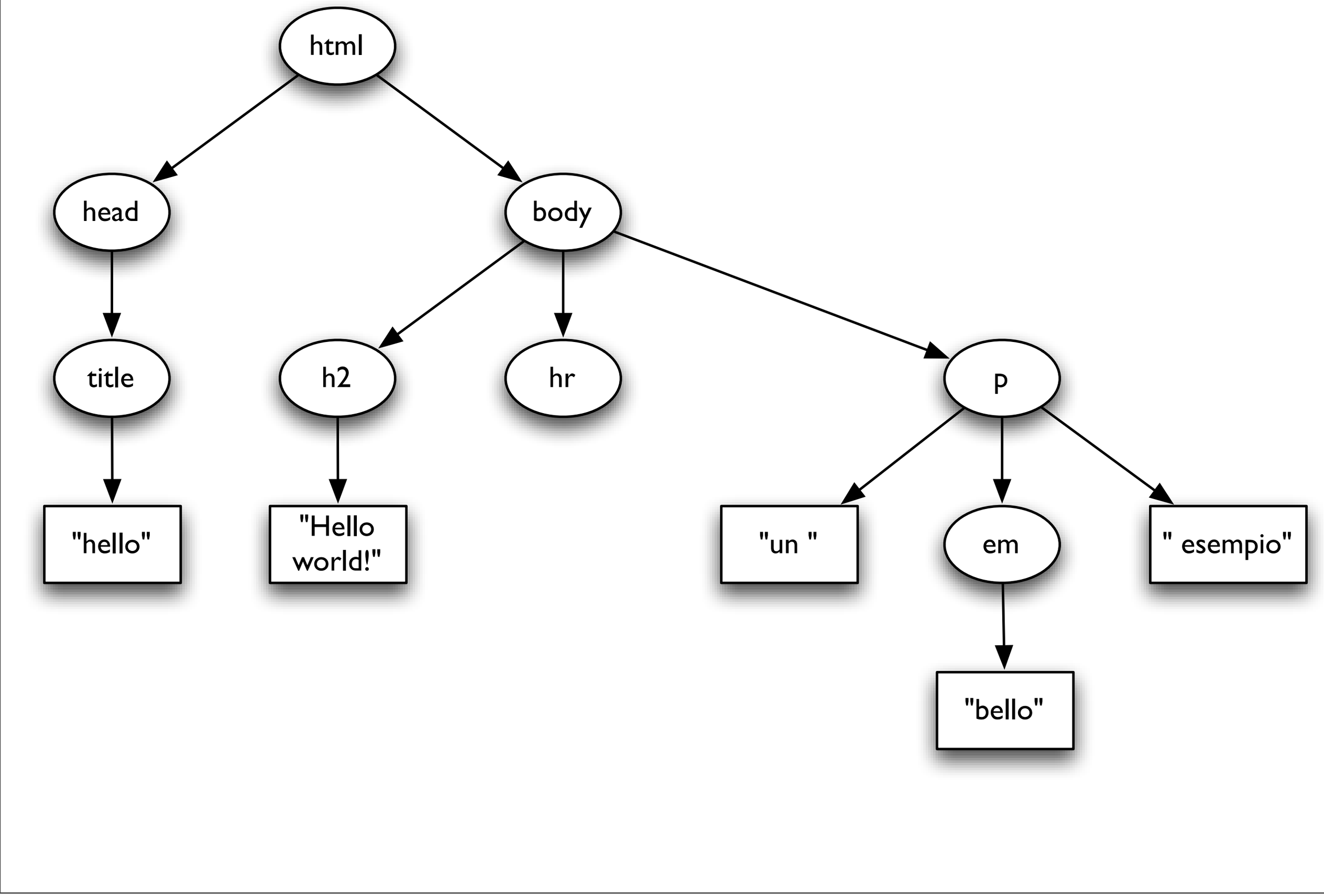


1×2+3×4

```
abstract class Expression {
    abstract double eval();
}

class Constant {
    ...
    double eval() {
        return value;
    }
}
```

```
class Plus {
    ...
    double eval() {
        return left.eval() + right.eval();
    }
}
```

Con la ovvia funzione di valutazione: 1x2+3x4 = 14!



Anche i document HTML sono una sorta di albero. Le foglie sono i “text nodes”. Il caso ricorsivo sono gli “elementi”, che possono contenere altri elementi e text nodes al loro interno.

```
public abstract class HtmlDocument {}
```

```
public class TextNode extends HtmlDocument {
    private final String text;

    public TextNode(String text) {
        this.text = text;
    }
}
```

```
new Element("p").add(new TextNode("Hello"));
//  --->  <p>Hello</p>

new Element("img").with("src", "/images/logo.png");
//  --->  <img src="/images/logo.png" />

new Element("a").with("href", "http://www.google.com").add(new TextNode("Google"));
//  --->  <a href='http://www.google.com'>Google</a>

public class Element extends HtmlDocument {
    private final String name;
    private List<HtmlDocument> contents = new ArrayList<HtmlDocument>();
    private Map<String, String> attributes = new TreeMap<String, String>();

    public Element(String name) {
        this.name = name;
    }

    public Element add(HtmlDocument content) {
        contents.add(content);
        return this;
    }

    public Element with(String attributeName, String attributeValue) {
        this.attributes .put(attributeName, attributeValue);
        return this;
    }
}
```

```java
public static Element paragraph(String text) {
    return new Element("p").add(new TextNode(text));
}

public static Element image(String source) {
    return new Element("img").with("src", source).with("alt", "");
}

public static Element link(String href, String label) {
    return new Element("a").with("href", href).add(new TextNode(text));
}

import static it.xpug.HtmlHelpers.*;

paragraph("Hello");
//  --->  <p>Hello</p>

image("/images/logo.png");
//  --->  <img src="/images/logo.png" alt="" />

link("http://www.google.com", "Google");
//  --->  <a href='http://www.google.com'>Google</a>
```

Si possono usare metodi statici per rendere più gradevole la sintassi della costruzione delle istanze di HtmlDocument. Si può usare lo “static import” per utilizzare questi metodi in qualsiasi classe.

```
HtmlDocument document =
    html(
        head(
            title("Hello builders!"),
            style("p { font-size: large; }")
        ),
        body(
            div(
                paragraph("foo"),
                paragraph("zork").with("class", "other"),
                link("http://www.google.com", "Google")
            )
        )
    );
```

Un esempio di come sia semplice costruire documenti HTML in Java usando gli helper statici.

```
String html =
    "<html>" +
    "<head>" +
    "<title>String concatenation sucks!</title>" +
    "</head>" +
    "<body>" +
    "<div>" +
    "<p>foo</p>" +
    "<p class=\"other\">zork</p>" +
    "<a href=\"http://google.com\">Google</a>" +
    "</div>" +
    "</body>" +
    "</html>";
```

Per contro, costruire html concatenando stringhe è una cosa molto poco conveniente.

```
abstract class HtmlDocument {
    abstract void renderOn(Writer writer) throws IOException;
}

class TextNode extends HtmlDocument {
    ...
    void renderOn(Writer writer) throws IOException {
        writer.write(text);
    }
}

class Element extends HtmlDocument {
    ...
    void renderOn(Writer writer) throws IOException {
        // simplified
        writer.write(String.format("<%s>", name));
        for (HtmlDocument node : contents) {
            node.renderOn(writer);
        }
        writer.write(String.format("</%s>", name));
    }
}
```

Quali funzioni ha senso definire su HtmlDocument? La più importante è una funzione che scrive su un Writer la rappresentazione testuale del documento HTML. Così potremo usarlo comodamente dentro a una servlet.

```
abstract class HtmlDocument {
    abstract Element findElementById(String id);
}
```

```
abstract class HtmlDocument {
    abstract Element findByNameAndClass(String name, String class);
}
```

```
abstract class HtmlDocument {
    abstract String textContent();
}
```

Altre funzioni utili: mi permettono di interrogare un HtmlDocument e fare *asserzioni*.

Come testare le view?

- Uguaglianza di stringhe
- Espressioni regolari
- XPath
- ...?

Test per uguaglianza di stringhe

```
@Test
public void willMakeAVerticalLayout() throws Exception {
    VerticalLayout layout = new VerticalLayout();
    layout.add(text("A"));
    layout.add(text("B"), text("C"));
    assertRenders(layout, "" +
            "<table>" +

            "<tr align='center'>" +
            "<td>A </td>" +
            "</tr>" +

            "<tr align='center'>" +
            "<td>B C </td>" +
            "</tr>" +

            "</table>");
}
```

Usare l'uguaglianza di stringhe non conviene

- Test fragile: basta aggiungere uno spazio e si rompe
- Test rigido: modifiche non significative (es. aggiungo una decorazione) rompono il test

Test con espressioni regolari

```
// Luca Marrocco, http://gist.github.com/309274

@Test
public void shouldArticleHtml() {
    Article article = new Article(mapmap(new String[]
        { "channel", "title", "testo", "oranotizia", "datanotizia" }));

    String html = new ArticleHtml(article, new Rules()).getHtml();

    assertThat(html, matchString("h2>.*title.*</h2"));
    assertThat(html, matchString("h3>.*channel.*</h3"));
    assertThat(html, matchString("h4>.*datanotizia.*</h4"));
    assertThat(html, matchString("p>.*testo.*</p"));
}
```

Le espressioni regolari sono un po’ meno rigide.

Test con XPath

```
@Test
public void betterTest() throws Exception {
    ...
    Document document = xmlDocumentFromString(writer.toString());
    assertEquals("Welcome!", getNodeContent(document, "/html/head/title"));
    assertEquals("green mouse",
        getNodeContent(document, "//a[@href='greenmouse.html']"));
}
```

```
"//a[@href='x']"
```

In XPath significa “un elemento *a* con *href*=‘*x*’

Con XPath posso fare asserzioni sulla struttura del documento xhtml in maniera flessibile

L’uguaglianza fra HtmlDocuments

- Non si rompe per modifiche di spazi bianchi

```
@Test
public void usesH1ToDisplayABigNumber() throws Exception {
    HexDisplay display = new HexDisplay(255);
    assertEquals(new Element("h1").add(new TextNode("0xff")), display);
}
```

Ricerche arbitrarie su HtmlDocument

- Possiamo scrivere test simili a quelli con XPath

```
Counter counter = new Counter("255");
HtmlDocument document = counter.toHtmlDocument();

Element inc = document.findLinkByLabel("inc");
assertEquals("C'è ma ha url errata", "?value=256", inc.getAttribute("href"));

assertEquals("0xff", document.findById("display").contentsAsText());
```