# Objects as hypertexts: how to render objects with HTML for teaching purposes

*Andrea Trentini, Daniela Micucci*
Andrea.Trentini@disco.unimib.it - Daniela.Micucci@disco.unimib.it
Dipartimento di Informatica Sistemistica e Comunicazione
Università di Milano-Bicocca
Via Bicocca degli Arcimboldi 8, MILANO, Italy
Tel. +39-02-64487856 Fax +39-02-64487839

**Abstract**

This is a description of a technique (and a tool, called HtmlStream) to visualize Java instances in HTML, Hypertext Markup Language (W3C 2000), format. It can be used to teach Java by clearly (and automatically) showing the relationships between class and instance and between classes and subclasses. Some basic knowledge of Java is required. This article is structured as following: 1) why we did it; 2) the output produced; 3) how to use it; 4) a consideration about UML, Unified Modeling Language (OMG 2000); 5) usage in actual courses; 6) final comments.

**Keywords:** object-orientation, visualization, class/instance, teaching, inheritance.

## 1. INTRODUCTION

We are currently working at CS Dept. of Milan University (Bicocca). Here, first year students are taught Java language. These students come from different high-level Italian schools and not everyone of them has previous experience with computer programming, alas very few. They have to face a lot of new basic concepts: first approach to programming, data types, flow control, etc. But they also have to face a lot of *advanced* (for them) object-oriented concepts: class/instance dualism, inheritance, binding, polymorphism and some basic UML syntax. Even the ones that come from technical high schools, where they usually have been taught (and should have learnt) at least a procedural language such as Pascal or C, have big problems in approaching an object-oriented language (even as simple as Java). Sometimes the *ones that know* have even more difficulties than the pure newcomers, because they try to map O-O concepts into procedural concepts (Decker 1993). We noticed that the two most difficult concepts, at least at the beginning of their object-oriented career, are:

- understanding the difference between class and instance: they keep declaring attributes and methods static, thus using Java as a procedural language (i.e. they don't create instances at all);
- understanding inheritance: they can't keep track of something they don't see (inherited attributes and methods) in a source file (in Java there is usually one class per file). Note that we make them use SDK by Sun (a command line environment for developing Java programs) instead of an IDE (Integrated Development Environment) with a class/object browser.

When introducing object-oriented concepts we usually draw *circlegrams* on a blackboard to show how an object is allocated in memory: of course this is a logical representation, not necessarily related with the actual implementation. In Figure 1 there is a common form of *circlegram*: the representation of an instance of class C with attributes correctly placed in their respective memory *ring* and with an example value for every attribute.

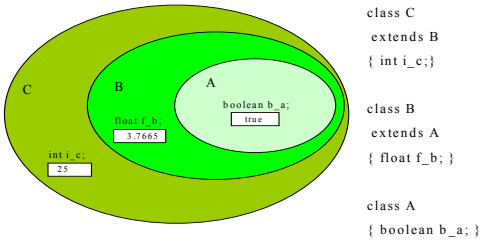


**Figure 1 – a circlegram**

What we needed was an implemented way of generating some kind of *circlegram* to let students use it inside their

programs. So we devised an HTML generator: a piece of code to create an HTML representation out of an instance in memory. We chose HTML to take advantage of all the HTML tools available on the net. At present, HtmlStream lets you *print an object* passing only the object reference to a special streamer object. This way you obtain HTML text that you can render with a browser (like Netscape or Internet Explorer). Some may ask why we did not choose XML (eXtensible Markup Language (W3C 2000)) as output format, but at the time we built our package we did not find any useful XML browser to effectively render that kind of output.

## 2. HTML REPRESENTATION

When you *print* an object using HtmlStream you get some HTML text structured this way:

| <**internal anchor**> <hashcode of object> <class> | |
|---|---|
| <name of field> | <value of field> |
| <name of field> | <value of field> |
| <name of field> | <if this field is a pointer, then this is a **link to an internal anchor**> |
| … | … |
| <if this objects extends another one, here you find a **"table in table" with the same format (recursive)**> | |

We used the HTML TABLE tag to represent an object instance, with a "recursive table in table" to represent inheritance. We hope to give the *circlegram* idea by directly including one *class level* into the other. For example, let's say that we have an instance of a class AnotherAgainMyClass, as drawn in Figure 2.

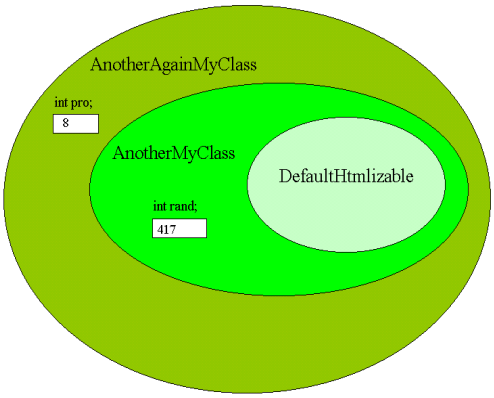


**Figure 2 – an instance described with a circlegram**

AnotherAgainMyClass inherits from AnotherMyClass that in turn inherits from DefaultHtmlizable. When converted into HTML by HtmlStream the result will be the one shown (already rendered) in Figure 3.

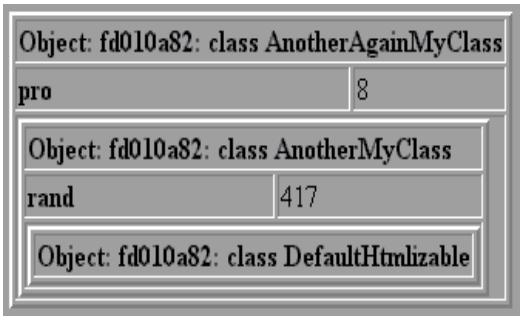


**Figure 3 – a complex object**

Another important feature of HtmlStream is that if you convert a graph of objects into HTML, you are in fact creating a large page with internal links. This happens when some class has *reference* attributes. In this case we use hyperlinks to link every attribute *value* (a hashcode) with its actual value, i.e. the referenced object. This way every *reference* attribute is represented as a hyperlink that, when clicked, brings you to the actual pointed object representation in the same page (see Figure 4). So that on the same page you can have more than a single object, all of them linked (as in memory) by clickable internal hyperlinks (anchors). This representation lets you interactively follow the actual referentiation between in-memory objects.

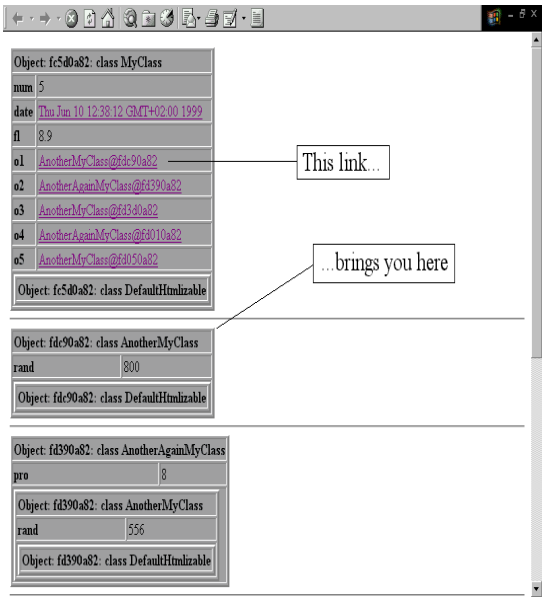


**Figure 4 - screenshot of an htmlized stream**

## 3. USING HTMLSTREAM

To use this package you must first download it (HtmlStream 2000). HtmlStream can be used only by instrumentating source code: to *htmlize* objects, you must put special *println* statements inside source code. When in need to *htmlize* an object instance you must follow this checklist:

- create an instance of an object called HtmlStream;

- pass the reference of the printable object to the HtmlStream, using the *grow()* method (see later in the examples);
- print the HtmlStream object in the usual Java way (*System.out.println…*), optionally redirecting the standard output to a file for later viewing; the output can also be piped to a command line HTML viewer, there are plenty on the Internet.

The only noticeable constraint for our user is that an object must be *printable* to be printable. In this htmlization environment an object is *printable* if it implements the HtmlizableI interface. That is because HtmlStream can work only on HtmlizableI objects. For this purpose we included in our package not only the HTML-converter (the HtmlStream class) but also a default implementation of a HtmlizableI class, called DefaultHtmlizable. Thus, any HtmlStream user can choose as preferred: a) implement from scratch his own HtmlizableI objects; b) inherit from DefaultHtmlizable.

An instance of HtmlStream, when passed an object to be htmlized, can automatically follow and convert every object reference inside the root one (the one passed initially), in a tree descending manner. This is very similar to the serialization mechanism available in Java where an object must be Serializable to be passed through an ObjectOutputStream and every link (except *transient* ones) is automatically followed.

**Some example code**

Here you can find a definition of an example class MyClass that extends DefaultHtmlizable (methods have been removed for clarity, they are present in the downloadable package file).

```
public class MyClass
         extends DefaultHtmlizable {
   protected int num;
   protected Date date;
   protected float fl;
   protected AnotherMyClass
       o1,o2,o3,o4,o5;
}
```

Suppose the following code placed somewhere inside a *main()* method: we create an instance, we pass it to the HtmlStream and then we print it.

```
/* MUST be a HtmlizableI object */
MyClass toBePrinted =
   new MyClass();

// "the printer"
HtmlStream hStream =
   new HtmlStream(toBePrinted);

// print the HTML representation
System.out.println(hStream);
```

The result should look like the one shown in Figure 5.

| Object: fc5d0a82: class MyClass | |
|---|---|
| num | 5 |
| date | Thu Jun 10 12:38:12 GMT+02:00 1999 |
| fl | 8.9 |
| o1 | AnotherMyClass@fdc90a82 |
| o2 | AnotherAgainMyClass@fd390a82 |
| o3 | AnotherMyClass@fd3d0a82 |
| o4 | AnotherAgainMyClass@fd010a82 |
| o5 | AnotherMyClass@fd050a82 |
| Object: fc5d0a82: class DefaultHtmlizable | |

**Figure 5 – *htmlized* object**

HtmlStream is also useful if you want to add other objects at a later time:

```
/* assume that we still have
   hStream in scope */

/* adding a new object */
AnotherHtmlizable
   anotherToBePrinted =
   new AnotherHtmlizable();

hStream.grow(anotherToBePrinted);

/* again adding a new object */
hStream.grow(
 new AgainAnotherHtmlizableI());

/* print the HTML representation,
   this time includes the
   new objects */
System.out.println(hStream);
```

The result is shown in Figure 4. This last example is one reason for HtmlStream class existence: it is useful when you need to queue more than one object for printing and there is no linking between them. A thinking aid when using HtmlStream this way is the "printing queue" metaphor: you submit objects to the stream (and the stream is able to follow every other referenced object), when you are satisfied you can start the actual printing of the stream itself.

## 4. UML PROXIMITY

Our students meet UML very early. We believe that our HTML format could help them grasp better the connections between class diagrams, source code and a runtime situation (a sort of *instance diagram*). The table representation should in fact be familiar to someone accustomed with UML class symbols. In our format inherited attributes are embedded inside the object instead of having an arrow that points to the extended

class. As an example take a look at the UML class diagram shown in Figure 6. This diagram is the actual class diagram of our example classes.

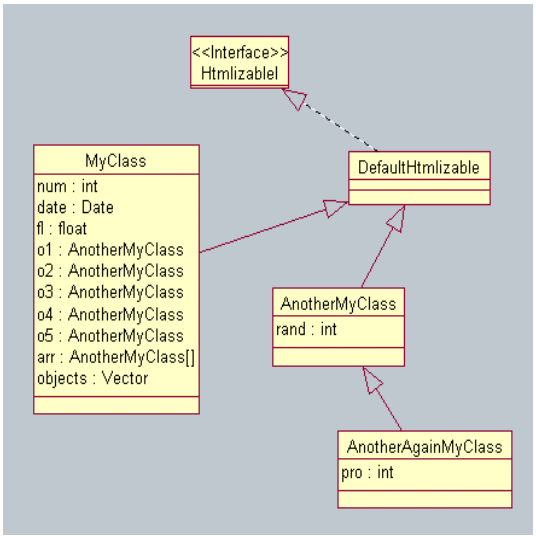


**Figure 6 - UML class diagram**

## 5. FEEDBACK ON USE

We tested our tool in two different learning environments: a University programming course and some business short Java courses.

At University we had a tough situation. As we mentioned at the beginning of this article, we have students coming from very different environments. The programming course is their first one, they have to learn how to login, how to edit and save files (knowing where they have saved... yes, we had students who almost lost their work because they didn't know where their files where!). They had to battle against operating system annoyances (e.g. not showing filenames completely i.e. extension hiding) and development environment (a command line one) syntax. Given this situation, we tried at first to explain them basic object-oriented concepts, but we reverted to a simpler procedural use of Java to let them absorb at least the basic programming concepts (variable types, scope, parameter passing, etc.).

In business courses life is somewhat easier. Learners usually come from other languages, at least they (should) know the basic principles of classic programming and they also know how to use a computer. In these environment HtmlStream has been useful, we succeded in *shifting* people from procedural to object-oriented programming. The most common misconception we could correct was the "state inheritance problem": programmers trying to access fields in an **instantiated** object by inheriting from that object class! At least we could easily show that class and object are not the same thing and that a class *has* many instances with different state.

## 6. CONCLUSION

The HtmlStream package lets you *graph* objects (even complex ones, recursively) by converting them to HTML text. This representation can be used to teach object-oriented basic concepts (class/instance, inheritance) by letting students print their own objects instead of (or complementary with) classroom blackboard drawings.

With HtmlStream there is no need to use a commercial/free IDE (Integrated Development Environment) with a graphical class browser or debugger. The only program needed is an HTML browser (nowadays omnipresent!). Students can thus use something they (should) know: an Internet browser (with the now common hypertext metaphor). Before creating HtmlStream we searched for ready-to-use tools for our purpose: the closest match was BlueJ (BlueJ 2000), but it doesn't show inheritance clearly in its object browser. Another close match is the JBuilder (Borland 2000) visual debugger that shows every variable in scope using a tree metaphor. The big problem with such a tree browser is that it doesn't avoid looping: if the user browses through a pointer to an already expanded branch, the branch will be expanded again, unbounded. Also, a lot of work about software visualization (both static and runtime) has been done ((Stasko 2000), (Domingue 1998) and (Mulholland 1998)), but we didn't find any object rendering mechanism as simple as ours.

We recently found an article (Dershem 1998) describing a tool very similar to our HtmlStream. The tool, called Object Visualizer, creates a GUI (Graphical User Interface) extracting properties from a given class using java reflection. Then the user can play with instances of that class by setting field values or calling methods. Our evaluation, based only on the article (the link to the Object Visualizer site was broken at the time of writing), is that HtmlStream may be better for educational purposes in the following aspects:

- it clearly shows every field at the respective class level while Object Visualizer makes you switch explicitly from one class level to the other
- it shows the complete state graph of an object (i.e. by following every reference to other objects)
- it also uses reflection, but HtmlStream shows **every** field (even private ones). We did it on purpose, to make students understand every detail of a class **they** are defining. Infact there is no purpose in information hiding if they have to understand what they (themselves) are doing...

Moreover, all the tools mentioned must be "graphically" used, they hide source code from the user/learner. We think that a novice should be kept very close to the language he is tryng to learn, even at the cost of a more steep learning curve (but in the right direction).

HtmlStream is written in Java, but the algorithm is very

simple. It can be ported to other object-oriented languages without great effort.

Finally, we built this tool/package only as a teaching aid. You can't use it as a production tool for visualizing objects, since you must extend a class to have HTML-printability, thus loosing the chance to extend from any other class (at least in Java).

## 7. REFERENCES


BlueJ, 2000, "Teaching oriented Java IDE", web site: http://www.sd.monash.edu.au/bluej/index.html

Borland, 2000, web site: http://www.borland.com

Decker R. and St. Hirsfield, 1993, "Top-Down Teaching: Object-Oriented Programming in CS1" ACM SIGCSE 1993, pp 270-273

Dershem, Herbert L. and James Vanderhyde, 1998, "Java Class Visualization for Teaching Object-Oriented Concepts", SIGSCE 98, pp. 53-57

Domingue, J. and P. Mulholland, 1998, "An effective web based Software Visualization learning environment", Journal of Visual Languages and Computing, 9 (5), pp. 485-508

HtmlStream, 2000, web site: http://www.sal.disco.unimib.it/~atrent/htmlstream.htm

Mulholland, P. and M. Eisenstadt, 1998, "Using Software to Teach Computer Programming: Past, Present and Future." In J. Stasko, J. Domingue, M. Brown and B. Price (Eds.), Software Visualization: Programming as a mutli-media experience. MIT Press: Cambridge, MA

OMG (Object Management Group), 2000, web site: http://www.omg.org

Stasko J., 2000, web site: http://www.cc.gatech.edu

W3C (WWW Consortium), 2000, web site: http://www.w3.org