

LCC 6310 The Computer as an Expressive Medium

Lecture 16

Overview

Programming concepts

HTML parsing, recursion

HTMLParser library from htmlparser.sourceforge.net

Parsing HTML

By parsing HTML you can process web information and create visualizations or manipulations of it

Last week we looked at the swing classes for HTML parsing

This week we'll learn a stand-alone Java package for HTML parsing

HTMLParser available from htmlparser.sourceforge.net

Go to Downloads/Current

Download the latest release build, release 1.6, June 10 2006

Before we start, a quick reminder about using external Java libraries within Processing...

Reminder: accessing external libraries

Place jars (or class files) within the **libraries** folder in processing

For example, if the class library is called **htmlparser**, create an **htmlparser/library** folder within the libraries folder and add the jar files

Use the **import** keyword to bring external classes into your program

Class libraries live in packages

`import <packagename>.*;` means "make all the classes in <packagename> available for use in my program"

Package names can be hierarchical (e.g. `org.htmlparser.util`)

If you get an error that a class can't be found, look in the documentation to see what package you need to import

The imported packages in the example code should get you pretty far

The Parser class

Parser is the main class for parsing the html file pointed to by a URL

What is parsing? To parse an html file means to turn the raw text of the page into structured tags that you can process

- Look for specific tags

- Get attributes from tags

One way to parse an HTML file with HTMLParser:

```
Parser parser = new Parser(<URL>);
NodeList nodelist = parser.parse(null);
```

`Parser.parse(NodeFilter filter)` returns a `NodeList` of all the HTML nodes (tags) that satisfy the filter

- The null filter returns a list containing all the tags

Reminder: Exceptions

Exceptions are thrown whenever java encounters an error situation

- You've probably all run into exceptions, like the `NullPointerException`

- Different kinds of exceptions are defined by Java, but programmers can define their own

When an exception is thrown, it travels up the call stack (the stack of method calls)

The default behavior for exceptions is for them to travel all the way to the top, where they terminate your program (and print out the exception)

To keep the program going, you can handle exceptions that might happen in your own code...

Reminder: Exceptions

To handle them, code that might cause certain kinds of exceptions should be enclosed by one of the following:

- A `try` statement that catches the exception

```
try {
    // some code that throws an exception
} catch (Exception e) {
    // what to do in case the exception happens
}
```

- A method that specifies that it can `throw` the exception

```
void myNastyMethod() throws Exception {
    // some code that throws an exception
}
```

We'll see that `Parser.parse()` throws an exception, so we'll need to handle it in our code

NodeLists

NodeLists contain `Nodes`, each representing a `tag`, `text` or `remark`

- Nodes are hierarchical, just like the structure of html documents

Let's look at the top-level tag structure for the class webpage:

```
println("There are "+nodelist.size()+" nodes...");
println("The tag nodes are:");
for (int i=0; i<nodelist.size(); i++) {
    Node n = nodelist.elementAt(i);
    if (n instanceof Tag) {
        Tag t = (Tag) n;
        println(t.getTagName());
    }
}
```

Let's see this in [Processing...](#)

Hierarchical structure

Since nodes are nested, our class webpage example contains 4 nodes, 2 of which are Tags:

```
<!DOCTYPE>
<HTML>
```

To get further content, we will need to examine the children of the top level nodes:

```
// we want to look at the children of the <HTML> tag
if (n instanceof Html) {
    NodeList sublist = n.getChildren();
    // look at the Tag nodes in the sublist...
}
```

Let's see this in [Processing...](#)

And also look at the [documentation...](#)

Quick aside: Iterators

We've seen how to step through all the objects in a list using a for loop and an integer counter. Another way to step through a list is to use a for loop with an **iterator**

Iterators are useful because they allow the caller to remove elements from the underlying collection during the iteration

The HTML parser package provides an iterator for a NodeList, called **SimpleNodeIterator**. The methods provided are:

boolean **hasMoreNodes()** - returns true if the iteration has more nodes

Node **nextNode()** - returns the next node in the iteration

We'll see how to use SimpleNodeIterator in a coming example...

Traversing the hierarchical structure

```
NodeList Node.getChildren()
```

Get a list of the children nodes of a node

So to search the nodeList for specific nodes, you need to search the top level, then search within the children of the top level, and so forth

```
NodeList NodeList.extractAllNodesThatMatch
(NodeFilter filter, boolean recursive)
```

If the second parameter is true, it will look in the children lists for you

Use a NodeFilter to tell it what kind of node you're looking for

Many kinds of node filters are already defined, such as TagNameFilter

Let's look at an example...

Example: find all images on a page

First create a **TagNameFilter** for the kind of tag you're looking for

```
TagNameFilter imgfilter = new TagNameFilter("IMG");
```

Use NodeList.**extractAllNodesThatMatch()** to create a list of just the image tags (make sure to set recursive searching to true)

```
NodeList imgnodes = nodelist.extractAllNodesThatMatch(imgfilter,true);
```

Let's see this in [Processing...](#)

Getting tag attributes

```
String Tag.getAttribute(String attribute)
```

Returns the String value of the attribute

Some examples of useful attributes

SRC for image tags (gives you the URL for the image)

ALT for image tags (gives you the alt text associated with an image)

HREF for link tags (gives you the URL associated with a link)

If a tag doesn't have the requested attribute, returns null

Let's try to draw the images from a webpage.

But first a quick reminder about... **Threads!**

Threads!

A **thread** is basically a program's path of execution

If a program runs as a single thread, then any time it needs to wait for something (e.g. to open a web page) it will lock up

To prevent this, you can make different parts of your program run in separate threads

So while one part of your program is waiting for something, other parts can still be doing things

In Java, every thread begins by executing a **run()** method in a particular object

run() is declared to be public, takes no arguments, has no return value, and is not allowed to throw any exceptions

Let's look at a simple example in [Processing...](#)

Example: Draw images from a webpage

Assume you have a NodeList of ImageTags (e.g. imgnodes)

```
for (SimpleNodeIterator i = imgnodes.elements(); i.hasMoreNodes();) {  
    ImageTag imgtag = (ImageTag) i.nextNode();  
    String imgurl = imgtag.getImageURL();  
    PImage pimg = loadImage(imgurl);  
    ...  
}
```

Let's look at a complete image drawer example in [Processing...](#)

Example: searching the alt attribute

Just like we used **getAttribute()** to get the image source, we can use it to get the image description text (the ALT attribute)

Use string methods to see if the alt text contains a search string

String.indexOf(<searchstr>) – if searchstr appears anywhere within the String, returns its location within the String, otherwise -1

```
boolean checkAltAtt(ImageTag imgtag, String searchstr) {  
    String altatt = imgtag.getAttribute("ALT");  
    if (altatt != null && !altatt.equals("")) {  
        if (altatt.indexOf(searchstr) != -1) {  
            return true;  
        }  
    }  
    return false;  
}
```

Let's see this in [Processing...](#)

Non-disturbing code

```
void method1() {
    println("I'm in method 1");
    method2();
}

void method2() {
    println("I'm in method 2");
}
```

Disturbing code

```
void method1() {
    println("Beginning of method 1");
    method1();
    println("End of method 1");
}
```

What will this do? It's **recursive**

Making the disturbing code work

```
void recursiveMethod(int depth) {
    if (depth <= 0) {
        return;
    } else {
        println("Beginning of method 1 at depth "+depth);
        method1(depth-1);
        println("End of method 1 at depth "+depth);
    }
}
```

Good recursive functions have a **base case** (where they stop) and a **recursive case** (where they call themselves)

Let's try this simple example in [Processing...](#)

Following links

When processing web information, we may want to crawl a site

This means following links and recursively parsing

Links are just another tag type, and the HREF is just an attribute...

Let's see an example...

Example: Following links recursively

```
if (depth > 0) { // crawl down to depth 0
  NodeList lnknodes = allnodes.extractAllNodesThatMatch(lnkfilter,true);
  if (lnknodes.size() == 0) return; // return if there are no links
  for (SimpleNodeIterator i = lnknodes.elements(); i.hasMoreNodes(); ) {
    LinkTag lnktag = (LinkTag) i.nextNode(); // get the link
    String href = lnktag.extractLink(); // returns an absolute link
    if (href != null && !href.equals("")) {
      if (!checkString(visited,href)) { // check if previously visited
        parseURL(href,depth-1); // recursively parse
      }
    }
  }
}
```

Let's look at the complete method in [Processing...](#)

Putting the pieces together

ImageCollage example in [Processing...](#)

Uses the following classes:

ImageCrawler.java

ImageDrawer.java

ImageObject.java

(make sure they are open in Processing)

Remember...

For **Thursday** this week: Theory Readings

One presenter (you know who you are!)

Everyone else: prepare one discussion question for the reading

Using Computers: A Direction for Design - Winograd & Flores (NMR pp. 551-561)