Intro to Project 1

Quick intro to pa1 (cont.)

- Trivial example, add this as Tokenizer’s main:

```java
token = new Tokenizer(new InputStreamReader(System.in));
while(token != null)
    System.out.println(token);
```

- Input:
  
  ```java
  hi this is fake Java
  // with comment syntax
  and "quoted stuff" /* and another comment */
  sothere
  ```

- Output:
  
  ```java
  hi
  this
  is
  fake
  and
  sothere
  ```

getNextID for a real Java source

- For a real Java source code (Xref.java), the ids found by getNextID() are underlined here:

```java
import java.util.Map;

public class Xref {
    public Xref(Reader inStream) {
        // Xref class interface: generate cross-reference

        /**
         * Class to perform cross reference
         * generation for Java programs.
         */
        public class Xref {
            // these are on line 1
            public Xref(Reader inStream) {
```

Notes on Tokenizer

- Note how both kinds of comments are skipped.
- When the Tokenizer sees // or /* it reads right through to end-of-line or */ without returning anything. Make sure you understand how it does this.
- The Tokenizer also keeps track of what line it is currently processing, so after you call getNextID() and get "import", you can call getLineNumber() and get its line number.
- A valid Java ID is defined (S&W pg. 11) as a sequence of letters, digits, underscores (_), and dollar-signs ($), the first of which is not a digit.
- However, the Java language spec says the $ character "should be used only in mechanically generated source code and only to access pre-existing names on legacy systems."
- The Tokenizer asks Java: Character.isJavaIdentifierPart(ch)
- Xref uses Tokenizer, in particular getNextID() and getLineNumber()
Example of Xref processing

```
1  import java.util;
2  // Xref class interface: generate cross-reference
3  /**
4   * Class to perform cross reference generation for Java programs.
5   * Input file (Xref.java)
6   */
7  public class Xref
8  {
9      public Xref(Reader inStream)
```

Building the Map: code examples

- To add “java” for the first time: it appears on line 1.
  ```java
  List<Integer> value = new ArrayList<Integer>();
  value.add(1);  // list with just line 1 in it
  theIdentifiers.put("java",value);  // "java" => (1)
  ```
- Add “public” the second time, on line 10:
  ```java
  List<Integer> lst = theIdentifiers.get("public");
  lst.add(10)
  // done, it's already in the Map! "public" => (7,10) now
  ```

Map Access examples

- When we get a ref on the List with get, we are obtaining the “live” object inside the Map, not a copy.
- Recall the examples in MoreOnSets slides showing elements dangling out of containers.
- So we don’t have to “put” the List back in the Map after changing it. We’ve already changed it in the Map.
- See next slide for picture…

Map.get returns a live list

```
10  get("public") return value: a ref to a live list inside the Map, ready for List.add, etc.
```

Using Maps

- What about equals/hashCode/compareTo here?
  - They are only needed for the domain type, here String, so the JDK has done all the work for us.
  - That is very commonly the case: we map from some sort of simple ID in the domain to a more complicated value in the range.
  - Could we use Map<String, Set<Integer>> here?
Using Maps of …

- Could we use Map<String, Set<Integer>> here?
- In general, sets don’t maintain order, so instead of
  Xref: 7, 10
  We might get Xref: 10, 7
- Unless we sorted the results at the end. The List maintains
  order for us.
- So the List is what we want here.