CSC 207 Lab 5

Purpose:
The purpose of this tutorial is to get you hands on training with HashTables in Java. We will replace code that contains multiple conditional statements (bad design) with something that uses a HashTable, resulting in a better design.

Intro:
I assume that you are familiar with python's *dictionaries.* In Java, HashTables are the same concept. (In fact, the class HashTable subclasses the (abstract) class Dictionary.)

Practical:
Important notice #1: In the following steps, don't follow the instructions blindly. It's a walkthrough, but it assumes you understand what and why you're doing it. The key is to make sure you understand and learn the ideas, techniques and tools involved. If you have trouble or are not sure: *ask the TA for clarification!*
Important notice #2: Your IDE gives you some pretty powerful tools: a compiler, code auto-completion, graphical indicators of errors, warnings, a debugger, etc. *Use them!* Save and compile your code as often as possible! Your code is not just text, it does stuff: *run it often!*

PART 1 (setting up, basically)

1. Download the java files from Blackboard. Create an empty Java project in Eclipse and load the starter code. You should know how to do that by now. If not, (a) boo, and (b) check last week's handout.

2. There are five classes in as many files: *file.java, openPDF.java, openWorld.java, openTxt.java* and *executionProgram.java.* Make sure you can compile and run the program as it is.

3. Checklist:
   a. Do you understand interfaces?
   b. Do you understand which classes implement the *File* interface?
   c. Do you understand why the classes that implement the interface must override the abstract functions from the *File* interface?
   d. Do you understand what polymorphism is and how it works?
      i. Do you understand how it works in the class level (at compile time)? In other words, how to extend a base class to add new functionality?
      ii. Do you understand what it means in terms of instances (at runtime)? In other words, how does a reference typed by the base class (the superclass) point to an object that is typed by the subclass?

   Important: The key here is to make sure you understand and learn these ideas. If you have trouble or are not sure: *ask the TA for clarification!*

PART 2

The amount of code that you will be writing here will be approximately 10-15 lines. What is most critical is that you understand how hash tables work.

We are going to write code that can open different file types, using a hashtable to look up which class contains the code for each file type.

1. **Before you start hacking away at the code read up the Javadoc documentation.**
   http://docs.oracle.com/javase/6/docs/api/java/lang/Class.html
   Basically, we can represent java classes themselves as *runtime objects*.
   This may be a bit tricky to really understand, so read carefully. So far you have seen examples where you had objects *typed* by some class. For example, the object *"some string"* has type that is defined by the class *String.*
   This is the usage of classes you have *already* seen. But the class *Class* is different!
   Just like you can get instances of the class *String* (such as "some string"), you can get instances of the class *Class*.
   For example, if you define:
   String x = "some string";
   Then the invocation x.getClass() returns an object of type *Class* that corresponds to the class *String.* Note that the method "getClass" is defined at the class *Object*, so you can call it on every java object.
   Read the Javadoc of *Class* for more information.
   If you're really curious you can read take a look here for even more information:
   http://en.wikipedia.org/wiki/Reflection_(computer_programming)

2. In the class *executionProgram*, write code that creates a Hashtable object. Then insert three items in it as shown in the table:

Key (this is string)	**Value (this is string)**
pdf	openPDF
doc	openDOC
txt	openTXT

   Read the Javadoc documentation for Hashtables here:
   http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html
   Most importantly, you need to understand the methods put() and get().

   Note: It is important that you match the value exactly as the name of the corresponding Class. For example, if your class is named *OPENPDF*, then the value for the key "pdf" should be the string "OPENPDF", ***not*** "openPDF".

3. Define a hard-coded string called:

   ```
   String fileExtension = “pdf”;
   ```

   We are cheating a bit here. If we were doing this for real, you would have to write code to automatically find the file extension (or more generally its MIMEtype) of whatever file we want to open. To do this you would need to write code that manipulates strings and extracts the appropriate substring that represents the extension. Since this particular lab’s focus is different, let’s just hard-code the value of *fileExtension* and pretend we got it from some string processing code.

4. Consider the scenario that your program is given a file to open. Assuming that we figured out the file’s *fileExtension*, we would have to create an appropriate “opener” object that would know how to open such files. For example, if we figured it was a pdf, then we would have to instantiate an object of type *openPDF* by calling the constructor of that class.
   Before you move further make sure you understand everything up to this point!!

5. But given that in reality the value of *fileExtension* would not be hard-coded, how would you know which class to instantiate (i.e., which constructor to call)?? A simple approach is to write if-statements:

   ```
   if (fileExtension.equals(“pdf”))
   {
           // call the openPDF constructor
   }
   else if (fileExtension.equals(“txt”))
   {
           // call the openTXT constructor
   }
   else if (fileExtension.equals(“doc”))
   {
           // call the openDOC constructor
   }
   else
   {
           // do some error handling if it was none of the known file types
   }
   ```

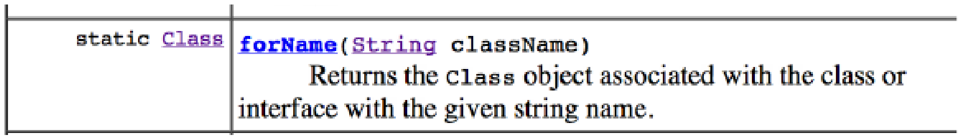
   The problem with this approach is that if you wanted to be able to handle 100 extensions, you would need to write 100 if statements!

6. Let’s try a different approach. Write some code that tells the Hashtable you defined in step2: “Look, I have this *fileExtension* string, use it as a *key* and get me the *value*”.
   You should be able to retrieve a string that corresponds to the name of the class that you

need to instantiate.

7. Now for the awesome part.
Look again at the Javadoc documentation for the class *Class* from step 1. Look for a method called *forName()*.
Read its documentation.
(By the way, is this an *instance method* or a *class method*? How are you going to invoke it? Do you need an instance of *Class*?)

8. You will use the *forName* method to get a class object.
Give it as an argument the string you got at step 6.

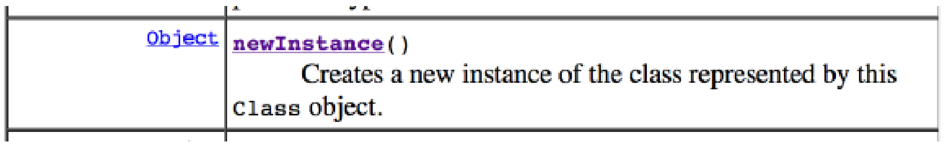


static Class	forName(String className) Returns the Class object associated with the class or interface with the given string name.

Do you understand the significance of the note at bottom of step 2?

9. You should now have a *Class* object that corresponds to your particular file extension. Cool, eh?

10. Go back to the *Class* Javadoc. Find the method called *newInstance().* Use this method to create a file opener instance for the appropriate class.



Object	newInstance() Creates a new instance of the class represented by this Class object.

11. See what you did there? With just two calls, starting from a boring, plain old string, you created a java object in memory.
http://tinyurl.com/kl5ncvo

12. Is it clear how to replace the design in step 5? What are the benefits of this new design as opposed to the if-statements?

13. If in the future you were asked to handle a new file extension (e.g. “.csc207”):
    a. How would this affect the original design?
    b. How would it affect the new design?