Object Oriented Programming with Java

Javadoc – CLI – Packages

Javadoc

JavaDoc: Overview

- Detailed description of code in a format that can generate spiffy API documentation, likely as HTML (just like what generated our API readings all semester long!)
- We can document **individual classes and their members** (directly in source code) **or entire packages** (through extra files).

JavaDoc Example: methods

```
/**
 * Short sentence summarizing: Returns a boolean
 * representing if the single input is prime.
 *
 * @param n the number to be tested if prime.
 * @return boolean for prime or not.
*/
public boolean isPrime (int n){
   //boring non-javadoc comment…
   for (int i=2; i<n;i++){
   if (n%i==0){ return false; }
   }
   return true;
}
```

JavaDoc Example: classes

```
/**
 * the Student class represents an individual
 * within the GMU community; each student has a
 * name, age, and studentID.
 *
 * @author George Mason
 * @version 0.10  April 1776
 */
public class Student {
 …
```

JavaDoc Example: packages

file: Foo/Bar/package-info.java

```
/**
 * the bar package provide various foo
 * interactions.
*/
package Foo.Bar;
//empty ever after
```

JavaDoc Notes

- We can write actual HTML in these comments.
- links can be created to other classes' documentation via **`{@link ClassName}`**
- There are various @tags that can be created: @author @version @param @return @exception @see @since @serial @deprecated

Generating Documentation: `javadoc`

- Use the javadoc command with various options:

```
javadoc -d /Users/me/htmlDir -subpackages Foo
```

- **-subpackages packageName** - please create documentation for this package and recursively through all subpackages as well. (convenient!)
- **-d some/Location/** - please put the (many) generated html files at some/Location that I've given you.
- **-classpath what:ever** – regardless of where I am, please use this classpath.
- **-sourcepath some/Location** – where can Javadoc look for files? Defaults to the classpath (convenient!)

More options: `javadoc`

- **-public, -protected, -package, -private** – please show anything as visible as what I've mentioned (use **-private** to show all; **-protected** is the default)
- **-exclude these:packages** - please don't include these packages or their subpackages, even though a **–subpackages** option refers to them.

- many more… perhaps you will learn on-the-job about them For now, we have enough to keep us occupied!

  Our goal is mostly to know what's out there so we can poke around, not to memorize/become experts at generating Javadocs.

Major javadoc tags

`@param` provides any useful description about a method's parameter or input it should expect

`@return` provides a description of what a method will or can return

`@see` will generate a link similar to the {@link} tag, but more in the context of a reference and not inline

`@since` specifies which version the class, field, or method was added to the project

`@version` specifies the version of the software, commonly used with %I% and %G% macros

`@throws` is used to further explain the cases the software would expect an exception

`@deprecated` gives an explanation of why code was deprecated, when it may have been deprecated, and what the alternatives are

Links Abound

- Much more info:

  http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

- Options for the **`javadoc`** command:

  https://www.baeldung.com/javadoc

  http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#runningjavadoc

- Doclets can be written to generate other outputs (such as pdfs, personalized/better(?) HTML, etc)

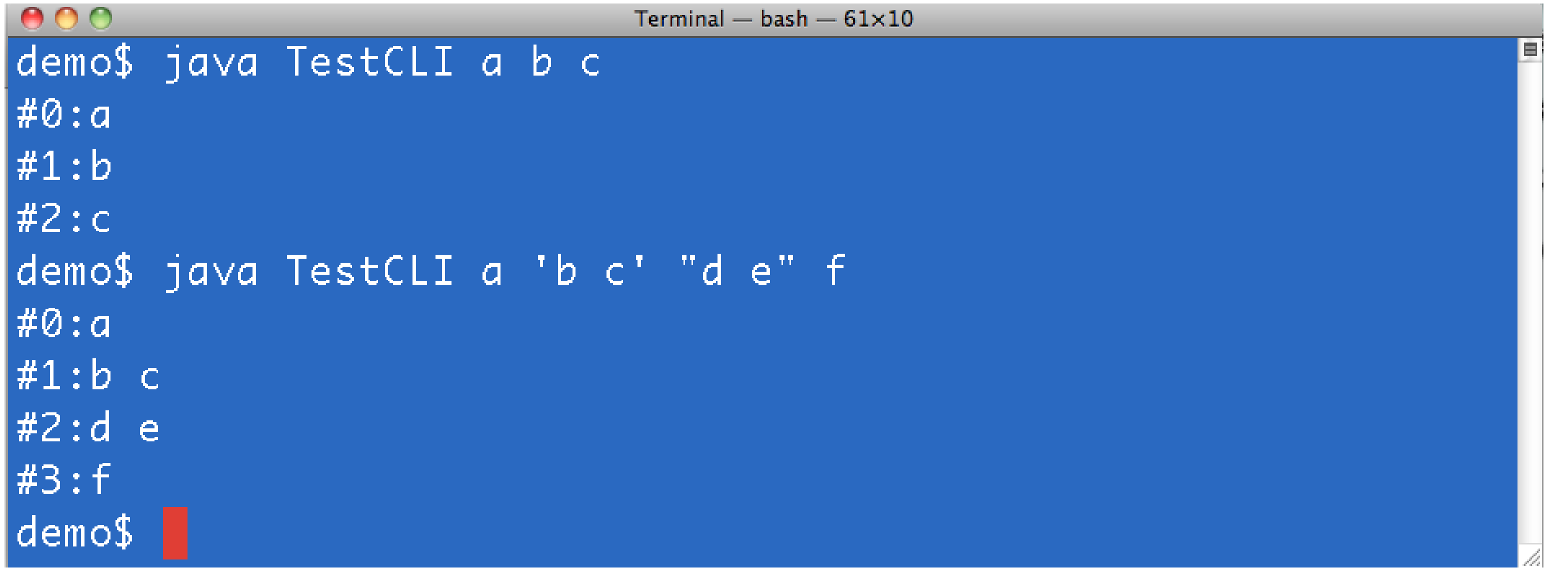
Command-line arguments

Command-Line Interface

- **Q:** Other than piping in input, how can we feed values to our compiled program from "the outside"?
- **Q:** What is that `String[]args` for, anyways?
- **A:** Command-line arguments!

CLI – Example

```java
public class TestCLI {
    public static void main (String[] args) {
        System.out.println("String[] args contained:");
        for (int i=0; i<args.length; i++) {
            System.out.println("#"+i+":"+args[i]);
        }
    }
}
```


Terminal — bash — 61×10

```
demo$ java TestCLI a b c
#0:a
#1:b
#2:c
demo$ java TestCLI a 'b c' "d e" f
#0:a
#1:b c
#2:d e
#3:f
demo$ 
```

CLI - Details

- Only String values are possible   `(String[] args)`

- Spaces separate values.
  - use single or double quotes to provide a single string value that contains spaces.

```
demo$ java Test one "two words" three "4 4 4"  'f i v e'
```

- Getting other types: call conversion methods.
  - Integer.java: public static int parseInt(String s) {…}
  - Double.java: public static double parseDouble(String s) {…}
  - (others, too)

Practice Problems

- Write a program that accepts command line arguments. If there were not exactly three arguments (which we will assume are double values), then print "invalid usage" and quit. If there were three, print "largest value of the three: ", and the actual largest value out of the three doubles that were passed in.

- Write a program that accepts an arbitrary number of integers on the command line; print out the sum, average, and maximum of those numbers.

- If you want to run these programs with different inputs, do you have to recompile between each run? Why or why not?

Three Versions of Input

We have three distinct ways we can get input for our program:

1. System.in: usually keyboard (but we can use < to pipe input from other places, like files)
2. Command-line arguments.
3. Directly use File-Reading within the program.

Three Versions of Input

- **System.in**: just a "stream of text" that your program can consume (for instance, through a Scanner).

  → Whether keyboard or piped input used, program is already compiled, choosing to consume this stream

- **Command-line args**: all Java programs' main methods have this.

  → If the (compiled) program relies on CL args, then different values can/must be given each execution.

- **Direct File I/O**: program is written specifically to access files, directly. No choice at execution time.

  → also, though, no CL args/streamed input is necessary.
  → how we get the filename can use any means though!

Packages

Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- The *Java standard class library* is part of any Java development environment
- It is provided by ~~Sun~~ Oracle and we might use it heavily
- Various classes we've already used (`System` , `Scanner`, `String`) are part of the Java standard class library
- Other class libraries can be obtained through third party vendors, or you can create them yourself

Class libraries

Why do we want to organize classes in libraries?

- Logical place to look for related classes
- Easy to reuse

Packages

- The classes of the Java standard class library are organized into *packages*
- Some of the packages in the standard class library are:

Package	**Purpose**
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

Package uses

- Packages are used to prevent naming conflicts
  → Imagine a large organization, with several people calling a class **`Account`**
- Packages must often be imported in to the current class
  - The compiler/runtime needs to know
    where to look for these imported classes

The import Declaration

- When you want to use a class from a package

  you could just use its *fully qualified name*, always

```
java.util.Scanner myScanner = new java.util.Scanner(System.in);
```

- Or you can **import** the class, and then use just the class name

```
//outside of class
import java.util.Scanner;

//inside a method of the class
Scanner myScanner = new Scanner(System.in);
```

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

NOTE: The fully qualified name of any class is the list of packages followed by the class name

The import Declaration

- All classes of the **`java.lang`** package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the **`System`**, **`Math`**, or **`String`** classes explicitly in earlier programs
- The **`Scanner`** class, on the other hand, is part of the **`java.util`** package, and therefore must be imported

How does Java know where to find these classes?

- Magic?...no
- The *path* to these classes must be set up on your machine and/or development environment
  - For example, through the environment variable PATH
  - The path points to a directory containing all the .jar (Java Archive) files
- You can then import any class contained in any .jar file specified in such a path

Importing classes

Example:

- you have Assert.class in the jar-file, junit-cs211.jar
- further, it's inside the org/ folder, and in the junit/ folder
- The Assert class is in the package **`org.junit`**
- You set your path to include the jar file, e.g.

```
-cp .:junit-cs211.jar
```

- You can then import the Person class

```
import org.junit.Assert;
```

Practical steps to create and run a package

1. Insert the **package** statement in your file

   e.g. `package mypack;`

2. Determine the package directory

   e.g. `javac -d . MyClass.java`

3. Run with the fully qualified name

   e.g. `java mypack.MyClass`