

Lecture 26: Design

Last time:

1. Project #5 is due Thursday 4/5 at 11 pm
2. Program design: algorithms, interfaces, use cases

Today:

1. Command-line Java
2. Code comments
3. Javadoc comments



Finish Project #5!

- Project due Thursday, April 4 at 11 pm
- Project is **closed**
 - You must complete the project by yourself
 - Assistance can only be provided by teaching assistants (TAs) and instructors
 - You must not look at other students' code
- Start now!
 - Read entire assignment from beginning to end before starting to code
 - Check out assignment now from CVS
 - Follow the instructions *exactly*, as much of grading is automated

Command-Line Java

- So far we have been using Eclipse to

- Create
- Run
- Debug
- Test

Java programs

- Is Eclipse the only way to write Java? No!
 - Any text editor can be used to create Java code
 - Java can be compiled and executed from the “command line” (= DOS prompt = Unix shell = etc.)

Creating Java Programs with a Text Editor



- You can use any text editor:
 - Wordpad
 - Notepad
 - Emacs
 - vi
 - etc.
- Be sure the file is saved as “plain text” (= ASCII)
- The name of the file must be `Foo.java` if the name of the class being defined is `Foo`

Compiling Java Outside Eclipse



- The Java compiler creates bytecode from Java source code
 - If the source file is `Foo.java` ...
 - ... the bytecode is contained in `Foo.class`
- How to execute the compiler outside of Eclipse?
 - Start a command window
 - Execute command `javac Foo.java`
 - This creates `Foo.class`
- Note: the operating system must be able to find `javac`!
 - My machine: `c:\Program Files\Java\jdk1.5.0_08\bin`
 - Windows use the notion of `path` = list of directories to locate commands
 - To add a directory to your path in Windows command window:
`set path= c:\Program Files\Java\jdk1.5.0_08\bin;%path%`

Running Java Code Outside Eclipse



- `java Foo`
 - Command `java` starts Java Virtual Machine
 - The JVM then executes bytecodes in `Foo.class`
- Command `java` is usually in same directory as `javac`
- `java` must be able to find `Foo.class`
 - Java uses a special path, `classpath`, for this
 - To add “current directory” (`.`) to `classpath`:
`set classpath=%classpath%;.`

Arguments to main

- Recall prototype of main method

```
public static void main (String[] args);
```

- args is array of Strings
- Where do these Strings come from?
- args come from operating system
 - When user runs executable ...
 - ... s/he can provide arguments

Example

- Consider `Foo.java`

```
public class Foo {
    public static void main (String[] args) {
        System.out.println("Hello world");
        // Print out number of arguments
        System.out.println("Number of arguments: " + args.length);
        // Print the args
        System.out.println ("Here are the args:");
        for (int i=0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

- Compiling `Foo.java` yields `Foo.class`

- Running

```
java Foo.class foo bar
```

yields following output:

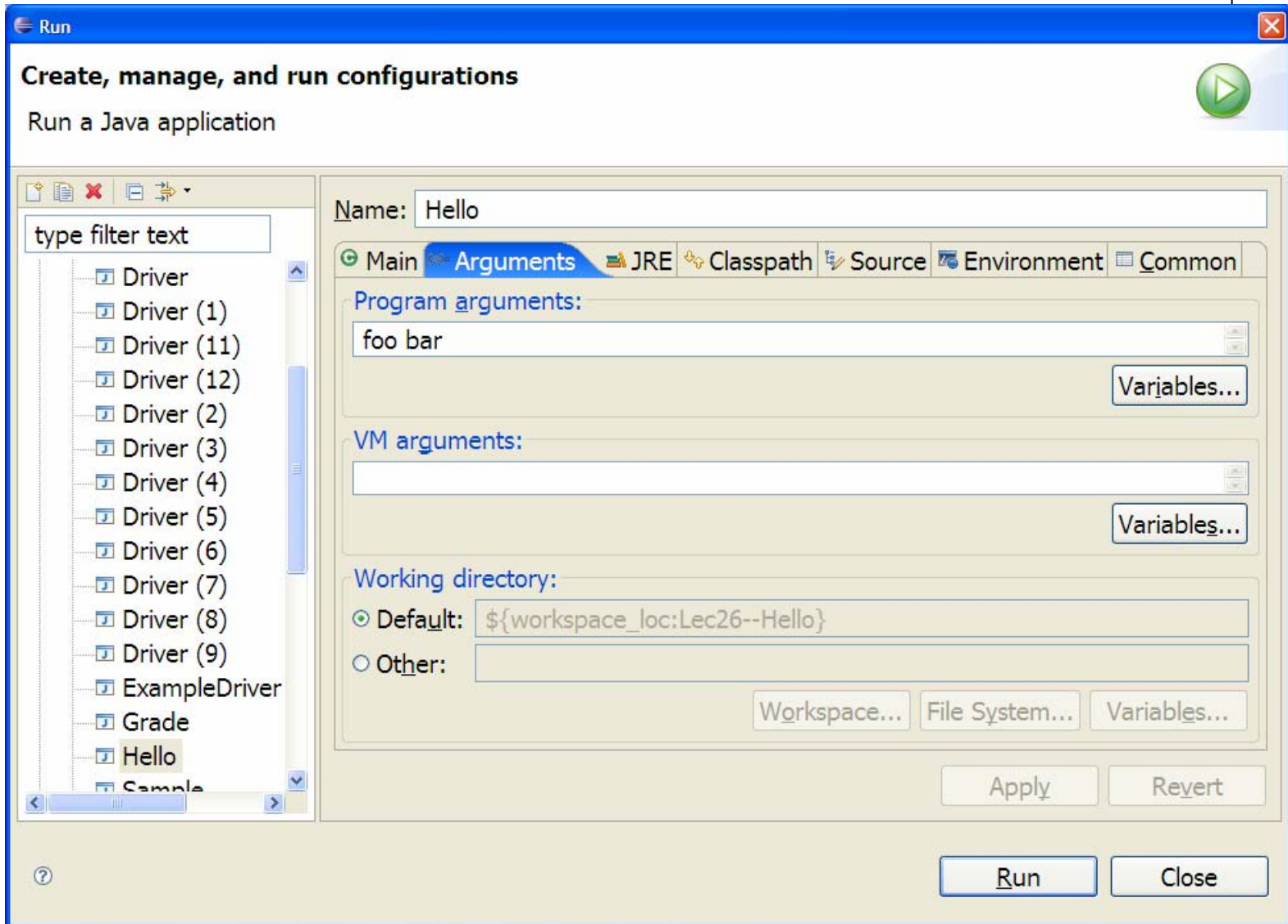
```
foo
bar
```

Providing Arguments in Eclipse



- Select Run → Run... from main Eclipse menu
- Make sure correct project, class specified
- Click on Arguments tab
- Type in argument in “Program arguments:” panel
- Run!

Example Run... Screen



Run

Create, manage, and run configurations

Run a Java application

Name: Hello

Main Arguments JRE Classpath Source Environment Common

Program arguments: foo bar Variables...

VM arguments: Variables...

Working directory:

Default: \${workspace_loc:Lec26--Hello}

Other:

Workspace... File System... Variables...

Apply Revert

Run Close

Back to Documentation!

- Programs are **hard to understand** (it's called *source code* for a reason)
- Good documentation is essential to making your program:
 - Usable
 - Maintainable
- Writing clear documentation is challenging, and requires careful thought

Types of Documentation

- Two uses for documentation
 - Usage: what does the software do, and how is it used?
 - Maintenance: how does the software do what it does?
- Two kinds of code commentary in Java
 - **Implementation comments**
 - Indicated by either `//` or `/* ... */`
 - `//` style comments are confined to a single line.
 - `/* ... */` block comments can span multiple lines.
 - Describes the (private/internal) coding and algorithm details
 - Usually appear interspersed throughout the code
 - **Intended audience: maintainers / reviewers** of the code
 - Should provide clarifying explanations and point out issues that are not obvious in reading the code (don't just repeat the code)
 - **Interface comments**
 - Indicated by `/** ... */`
 - Describes information about class / method, including:
 - Author
 - Parameters and their meaning
 - Return value (if any)
 - Etc.
 - Should appear at the top of method/class definition
 - **Intended audience: users** of the code

Common Errors in Commenting



Too many comments: This can obscure the flow of your program.

Too few comments: Your intent may not be understood (code is never “self-documenting”!)

Comments that repeat the code:

```
int total = 0;    // initialize integer total to 0
```

Using comments to conceal unclear code: ...just rewrite the code

```
double a = h*w;    // set the area (a) to the height (h)
                  // times width (w)
```

```
double area = height * width;
```

Uninformative comments: “What the heck does that mean?”

```
double d = processValue( );    // Change later (legacy)
```

Misleading / erroneous comment: These are dangerous

```
for ( int i = 0; i < a.length-1; i++)    // run through the
                                          //whole array
```

(Actually it misses the last element. Was this intended or an error?)

Javadoc

- A tool for converting interface documentation into HTML
 - HTML = “Hypertext Markup Language”
 - Used for creating web pages
 - Javadoc:
 - Reads your source code
 - Produces formatted documentation in HTML
- How to run Javadoc in Eclipse
 - Right-click on the project name and select “Export→Javadoc”
 - Enter Javadoc command if necessary (javadoc.exe on Windows, javadoc in Unix / Linux)
- What Javadoc does
 - Extracts the declarations of your public methods and public instance variables, from your classes and interfaces
 - Extracts the contents of interface comments (ones that start with /**)
 - Creates navigable web-pages containing this info

Javadoc Documentation

- **Class comments:** Immediately prior to each public class, add a javadoc comment that **explains what the class does**. You can also add the following special “**tags**”, which javadoc recognizes and provides special formatting for:

@author – the author of the class

@version – the current software version number

@see – refer the reader to related classes

- **Example:** In Rational.java

Sample javadoc
class comment

```
/**
 * This class implements a rational number object,
 * and provides methods for performing arithmetic
 * on rational numbers.
 * @see java.lang.Math
 * @author Schultzie von Wienerschnitzel III
 * @version 3.14159
 */
public class Rational { ... }
```



Sample Javadoc Output

The screenshot shows a Mozilla browser window titled "Rational - Mozilla". The address bar is empty. The menu bar includes File, Edit, View, Go, Bookmarks, Tools, Window, and Help. The main content area displays the Javadoc for the `Rational` class. On the left, there is a sidebar titled "All Classes" with links to `Driver`, `PrimeGenerator`, and `Rational`. The main content area has a navigation bar with links for "Package", "Class", "Use", "Tree", "Deprecated", "Index", and "Help". Below this, there are links for "PREV CLASS", "NEXT CLASS", "FRAMES", "NO FRAMES", "SUMMARY: NESTED | FIELD | CONSTR | METHOD", and "DETAIL: FIELD | CONSTR | METHOD". The main content area displays the following Javadoc output:

```
Class Rational

java.lang.Object
└ Rational

public class Rational
extends java.lang.Object

This class implements a rational number object, and provides methods for performing
arithmetic on rational numbers.

Version:
    3.14159

Author:
    Schultzie von Wienerschnitzel III

See Also:
    Math
```

Javadoc Documentation

- **Method comments:** Immediately prior to each public method, add a javadoc comment **explaining what the method does**, the meanings of the **parameters**, the **return value**, and any **errors**. The following tags are recognized:

@param – give the name and description of each parameter. There should be one for each parameter.

@return – describe the return value (unless it is void)

@throws – (**Later**: we will discuss error exceptions later this semester)

@deprecated – (**Usually for system use**: indicates that a method should be avoided, since better alternatives exist)

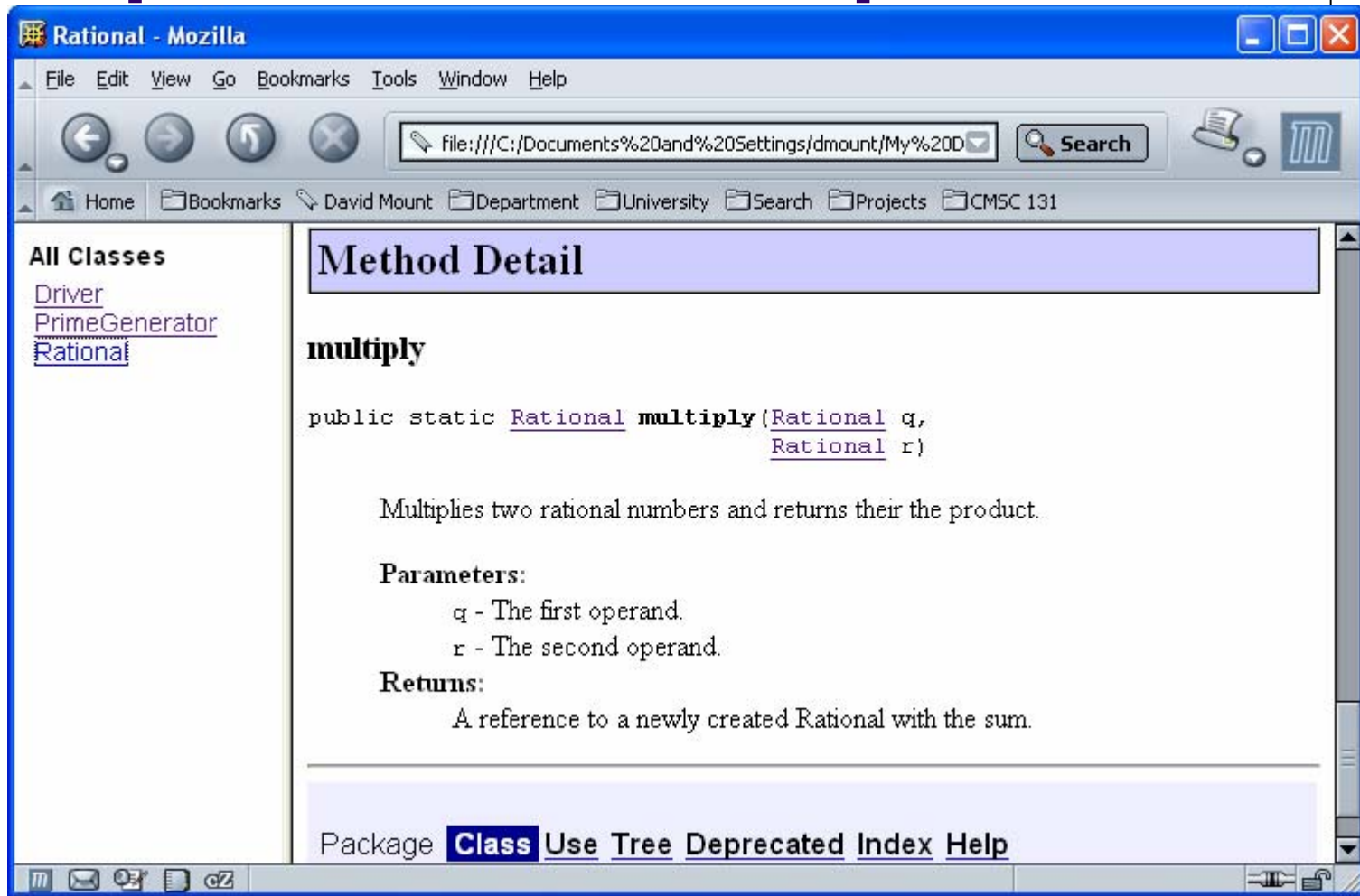
- **Example:**

```
/**  
 * Multiplies two rational numbers and returns their the product.  
 * @param q The first operand.  
 * @param r The second operand.  
 * @return A reference to a newly created Rational with the sum.  
 */
```

```
public static Rational multiply( Rational q, Rational r) { ... }
```

Sample javadoc method comment

Sample Javadoc Output



The screenshot shows a Mozilla browser window titled "Rational - Mozilla". The address bar contains the file path: `file:///C:/Documents%20and%20Settings/dmount/My%20D`. The browser displays a Javadoc page for the `multiply` method of the `Rational` class. On the left, a sidebar titled "All Classes" lists `Driver`, `PrimeGenerator`, and `Rational`. The main content area is titled "Method Detail" and shows the following information:

```
multiply  
  
public static Rational multiply(Rational q,  
                                   Rational r)
```

Multiplies two rational numbers and returns their the product.

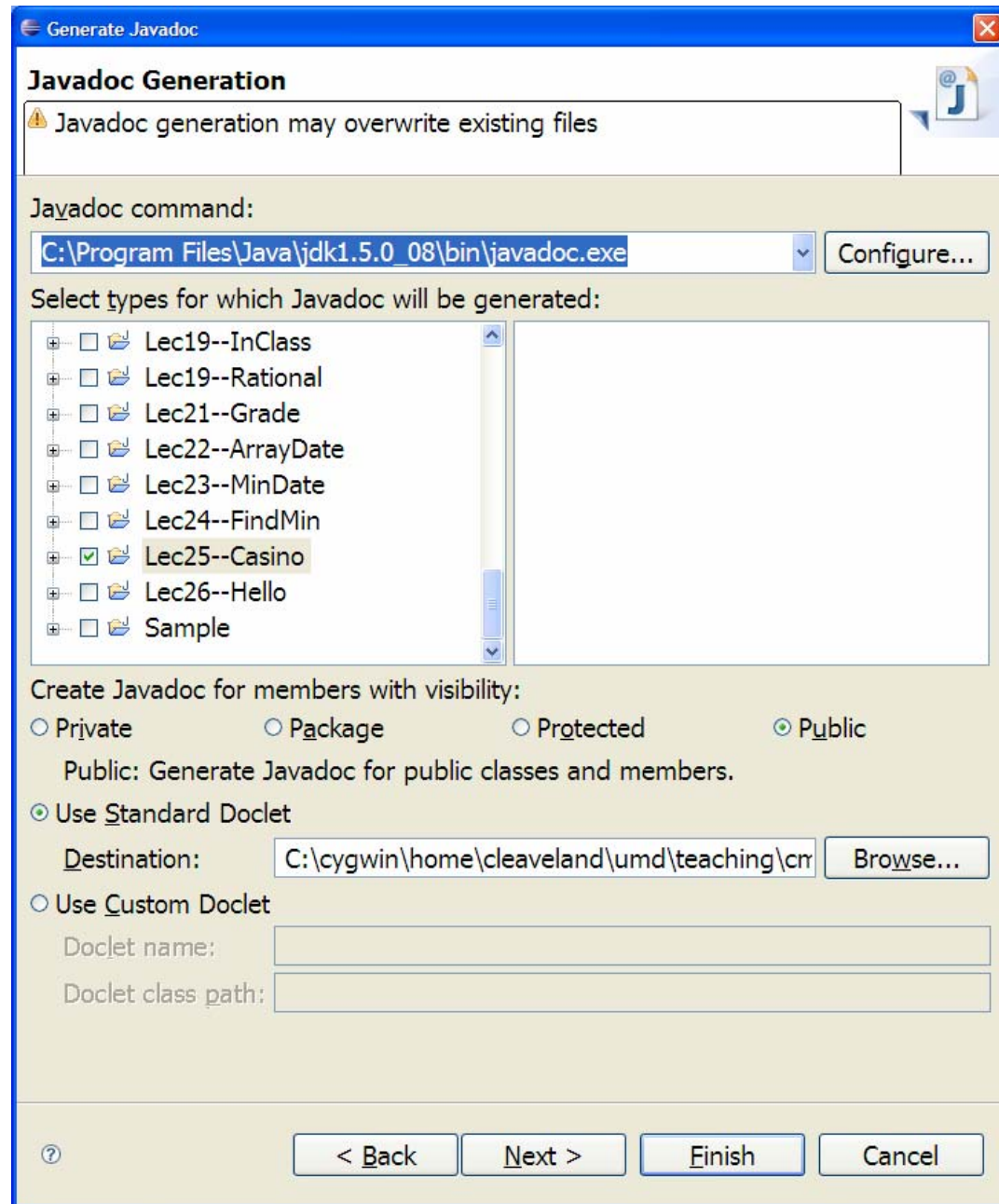
Parameters:
q - The first operand.
r - The second operand.

Returns:
A reference to a newly created Rational with the sum.

At the bottom of the page, there is a navigation bar with the following links: Package, **Class**, Use Tree, Deprecated, Index, Help.

Launching Javadoc

- Right-click on the project name and select “Export→Javadoc”
- Enter Javadoc command if necessary (javadoc.exe on Windows, javadoc in Unix / Linux)



Where Does Javadoc Put Its Output?



- By default, Javadoc creates folder named “doc” in a project
- It also creates a file “index.html” inside doc
- index.html is the main access point for the documentation for the project

Two Ways to View Javadoc Output



- Javadoc produces web pages
- To view the pages, you use a browser
- One possibility: Eclipse
 - Eclipse includes a browser
 - The Javadoc files are shown in the Eclipse navigation panel
 - You can open the documentation files from the panel
- Another possibility: an external browser
 - Open browser
 - Use “browse” feature to find folder containing documentation

Interface Documentation: Classes



- Immediately before public class, add a Javadoc comment block
- Block should contain
 - Brief explanation of purpose of class
 - Author
E.g. `@author Fawzi Emad`
 - Date (via version)
E.g. `@version 4/4/2007`
- Other useful information
 - **@see** – refer the reader to related classes

Interface Documentation: Methods



- Immediately prior to each public method, add a Javadoc comment block
- Block should contain
 - Brief explanation of method's functionality
 - The meanings of the parameters, using `@param` tags
 - The meaning of the return value, using the `@return` tag
 - The exceptions thrown (later), using the `@throws` tag
 - Whether or not this method is superseded by others, using the `@deprecated` tag
- Example: answer method in answerQuestion method

```
/**  
 * Determines the answer to any question  
 * @param question The question that is being answered  
 */  
public String answerQuestion(String question);
```

Javadoc Output



All Classes

[CanHost20QuestionsGame](#)
[Dog](#)
[Driver](#)
[Foo](#)
[GrandOracleOfWisdom](#)
[Politician](#)
[TwentyQuestions](#)

Method Summary

<code>java.lang.String</code>	answerQuestion (<code>java.lang.String question</code>) Determines the answer to any question
<code>java.lang.String</code>	getName () Getter for name
<code>java.lang.String []</code>	getTauntArray () Provide an array of funny things that could be said to "taunt" someone.

Method Detail

getName

`java.lang.String` **getName**()

Getter for name

answerQuestion

`java.lang.String` **answerQuestion**(`java.lang.String question`)

Determines the answer to any question

Parameters:

`question` - the question that is being answered

getTauntArray

`java.lang.String []` **getTauntArray**()

Provide an array of funny things that could be said to "taunt" someone.

Returns: