CMSC 131
Object-Oriented Programming I

Javadoc

Dept of Computer Science
University of Maryland College Park

This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane
Overview

- Javadoc
Code Coverage

- What is code coverage?
- What code coverage information is provided by the submit server?
Overview: We have already seen how to write comments.
- Programs are **hard to understand**, and this can result in **errors**.
- Good documentation is essential to making your program **easy to use and maintain**.
- Writing **clear documentation** is like writing **clear prose**. It is **challenging**, and requires careful thought.

Syntactic Rules and Conventions:
- // style comments are confined to a single line.
  - Are best used to explain the meaning of a **single line** of code.
  - Provide (private) **implementation details** for a programmer who may be modifying your code.
- /* ... */ block comments can span multiple lines.
  - Are best used to explain the meaning of a **block** of code.
  - Provide both (private) **implementation details** as well as (public) **interface information**.
Principal Types of Documentation

- **Documentation Comments (Javadoc):** Indicated by /** ... */
  - Describes the (public) behavior the method, its parameters and their meaning, the return value (if any), and possible errors or exceptions. Should appear at the top of method/class definition
  - To be read by a user of your method or class.

- **Implementation Comments:** Indicated by either // or /* ... */
  - Describes the (private/internal) coding and algorithm details.
  - Usually appear interspersed throughout the code. Help you understand what the code does
  - To be read by someone programming/modifying the method
  - These comments should not duplicate the code. Rather, they should provide clarifying explanations and point out issues that are not obvious in reading the code
Javadoc Documentation

- **Javadoc:**
  - Reads your source code and produces formatted documentation as an **HTML file**, which can be viewed in a **web browser**
  - Allow us to keep code and documentation together (big advantage)

- **How it works:**
  - To run it from Eclipse, right-click on the project name and select **“Export→Javadoc”**.
  - **javadoc is a program** (javadoc on Unix/Linux and javadoc.exe on Windows). It scans all the files in your project directory.
  - It extracts the **declarations** of your **public methods** and **public instance variables**, from your classes and interfaces.
  - It extracts the contents of **block comments** that start with “/**”. Example:
```java
/**
 * This is a javadoc comment.
 */
```
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

  ```java
  /**
   * 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 { ... }
  ```
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` → exceptions thrown
  - `@deprecated` → *(Usually for system use)*: indicates that a method should be avoided, since better alternatives exist

- **Example**:

```java
/**
 * 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) { ... }
```
Method Detail

multiply

public static Rational multiply(Rational q, Rational r)

Multiplies two rational numbers and returns their product.

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

Returns:
A reference to a newly created Rational with the sum.
Commenting Examples: Consider a method that computes all the **prime numbers** from 2 up to a given value **maxNumber**

Prime: A number $p > 1$ is **prime** if it is divisible only by itself and 1

Method: Sieve of Eratosthenes:
- **List** all the numbers from 2 up to **maxNumber**
- For each number, **remove** all its larger **multiples** (set to 0)
- **Stop** when reaching the **square root** of **maxNumber**
Example: Prime Number Generator

- **Method:** Sieve of Eratosthenes: (For maxNumber = 20)
  - **List** all the numbers from 2 up to maxNumber.
  - For each number, **remove** all its larger multiples (set to 0).
  - **Stop** when reaching the square root of maxNumber.

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
</table>

2  3  0  5  0  7  0  9  0  11  0  13  0  15  0  17  0  19  0

5 > sqrt(20) and so we are done.

Final primes:

2  3  5  7  11  13  17  19
Implementation Issues:

Array bounds: We want to store values ranging from from 2 up to maxNumber. To do this, we will declare the array to have size maxNumber+1. Thus the indices run from [0..maxNumber], but we will simply not use entries 0 and 1.

When to stop? Clearly we could repeat the procedure for all primes up to maxNumber, but this would not be efficient. Any nonprime number is eliminated by its smallest prime divisor.

14 = 2 * 7 will be eliminated by 2
195 = 3 * 5 * 13: will be eliminated by 3
289 = 17 * 17 will be eliminated by 17

We do not need to search beyond the square root of maxNumber, since if it hasn’t been eliminated by then, it never will be
This class demonstrates a clear and simple use of comments with a single method that generates a list of primes. It also provides an example of how JavaDoc documentation works.

@version 1.0

public class PrimeGenerator {

/**
 * Returns an array containing the prime numbers between 2 and the given parameter. If there are no primes found, an array of length 0 is returned.
 * @param maxNumber The upper bound on the range of primes.
 * @return An integer array holding the prime numbers.
 */

public static int[] getPrimes( int maxNumber ) {
    // ... (continued below)
}
public PrimeGenerator()

Method Detail

getPrimes

class getPrimes(int maxNumber)

Returns an array containing the prime numbers between 2 and the given parameter. If there are no primes found, an array of length 0 is returned.

Parameters:
maxNumber - The upper bound on the range of primes.

Returns:
An integer array holding the prime numbers.
public static int[] getPrimes(int maxNumber) {

    /* This is based on the sieve of Eratosthenes. The array values[ ] contains the values
     * from 2 up to maxNumber. Each nonzero value is used to eliminate all its larger multiples. */

    int[] values = new int[maxNumber + 1]; // array of values ranging from 2 to maxNumber
    for (int i = 2; i <= maxNumber; i++) values[i] = i; // initialize values starting at 2

    /* Compute the primes by removing (zeroing) multiples of primes. */
    for (int i = 2; i <= (int) Math.sqrt(maxNumber); i++) {
        for (int j = 2 * i; j <= maxNumber; j += i) values[j] = 0;
    }

    /* Count the number of remaining primes */
    int nPrimes = 0;
    for (int i = 2; i <= maxNumber; i++)
        if (values[i] != 0) nPrimes++;

    /* Copy the primes to the result array */
    int[] primes = new int[nPrimes];
    int j = 0;
    for (int i = 2; i <= maxNumber; i++)
        if (values[i] != 0) primes[j++] = values[i];
    return primes;
}

class PrimeGenerator (Part 2)

This initial block comment explains (to a programmer) how we implemented it.

Add smaller block comments to explain what each major section of the code does.

Note that comments within the method are directed towards someone programming/modifying the method, as opposed to a user of the class.