Documenting, Using, and Testing Utility Classes

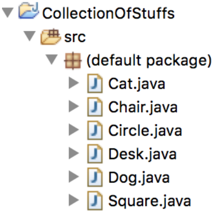
Readings: Chapter 2 of the Course Notes

EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

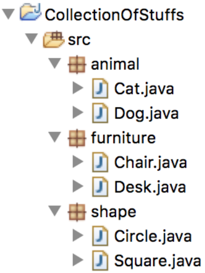
Structure of Project: Packages and Classes

A Java *project* may store a list of Java *classes*.



```
▼ CollectionOfStuffs
  ▼ src
    ▼ (default package)
      ▶ Cat.java
      ▶ Chair.java
      ▶ Circle.java
      ▶ Desk.java
      ▶ Dog.java
      ▶ Square.java
```

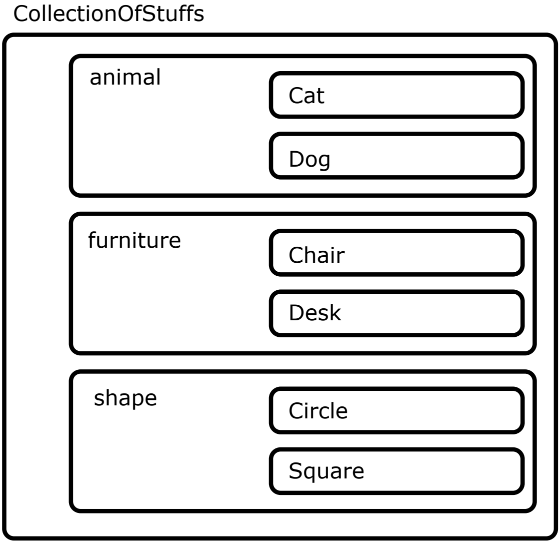
You may group each list of related classes into a *package*.



```
▼ CollectionOfStuffs
  ▼ src
    ▼ animal
      ▶ Cat.java
      ▶ Dog.java
    ▼ furniture
      ▶ Chair.java
      ▶ Desk.java
    ▼ shape
      ▶ Circle.java
      ▶ Square.java
```

To see project structure in Eclipse: `Package Explorer` view.

Visibility: Project, Packages, Classes



Visibility of Classes

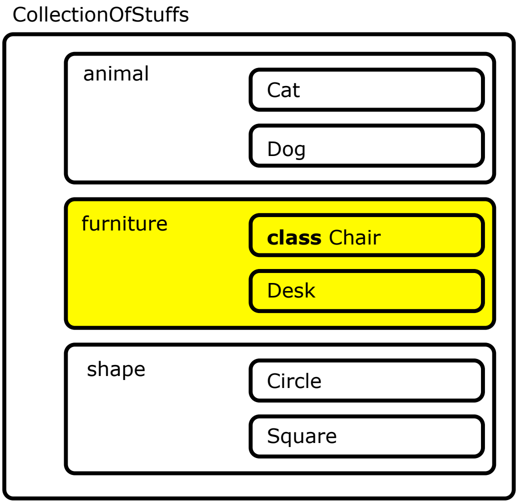
- Only one modifier for declaring visibility of classes: *public*.
- Use of `private` is forbidden for declaring a class.

  e.g., `private class Chair` is **not** allowed!!
- **Visibility** of a class may be declared using a modifier, indicating that it is accessible:
  1. Across classes within its resident package [no modifier]

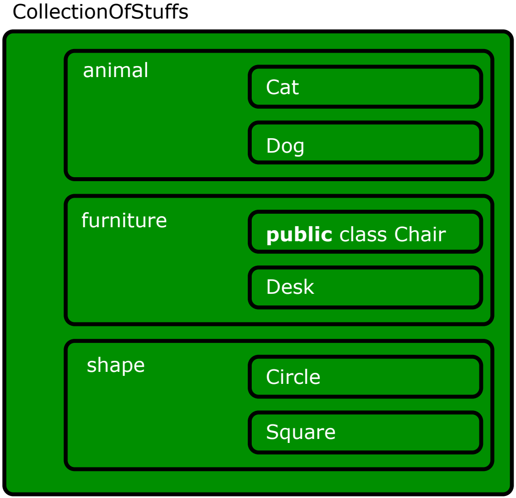
     e.g., Declare `class Chair { ... }`
  2. Across packages [*public*]

     e.g., Declare `public class Chair { ... }`
- Consider class `Chair` in: Resident package `furniture`; Resident project `CollectionOfStuffs`.

Visibility of Classes: Across All Classes Within the Resident Package (no modifier)



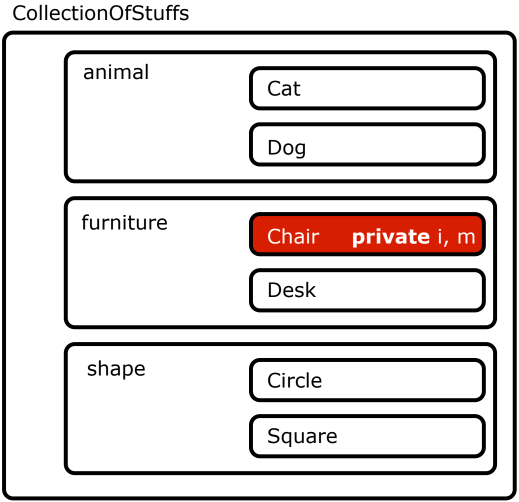
Visibility of Classes: Across All Classes Within the Resident Package (no modifier)



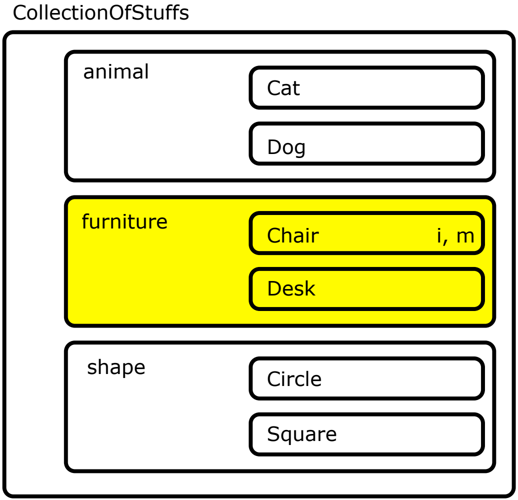
Visibility of Attributes/Methods: Using Modifiers to Define Scopes

- Two modifiers for declaring visibility of attributes/methods: *public* and *private*
- **Visibility** of an attribute or a method may be declared using a modifier, indicating that it is accessible:
  1. Within its resident class (*most* restrictive) [*private*]
     e.g., Declare attribute `private static int i;`
     e.g., Declare method `private static void m(){};`
  2. Across classes within its resident package [no modifier]
     e.g., Declare attribute `static int i;`
     e.g., Declare method `static void m(){};`
  3. Across packages (*least* restrictive) [*public*]
     e.g., Declare attribute `public static int i;`
     e.g., Declare method `public static void m(){};`
- Consider `i` and `m` in: Resident class `Chair`; Resident package `furniture`; Resident project `CollectionOfStuffs`.

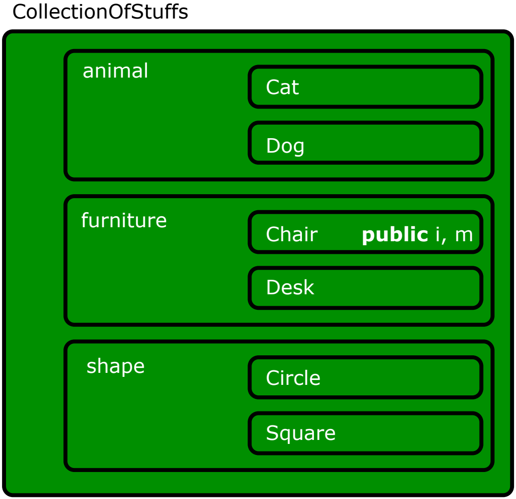
Visibility of Attr./Meth.: Across All Methods Within the Resident Class (`private`)



Visibility of Attr./Meth.: Across All Classes Within the Resident Package (no modifier)



Visibility of Attr./Meth.: Across All Packages Within the Resident Project (`public`)



Structure of Utility Classes

- *Utility classes* are a special kind of classes, where:
  - All *attributes* (i.e., stored data) are declared as *static*.
  - All *methods* (i.e., stored operations) are declared as *static*.
- For now, understand all these *static* attributes and methods collectively make their resident utility class a **single** (i.e., one that cannot be duplicated) machine, upon which you may:
  - Access the value of a data item. [attribute]
  - Compute and return a value. [accessor]
  - Computer and change the data (without returning). [mutator]
- We will later discuss non-static attributes and methods.

To see class structure in Eclipse: `Outline` view.

Structure of Utility Classes: Example (1.1)

```java
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Three independent groups of modifiers in the above utility class:

1. Access :*private* (**L2**, **L13**), *public* (**L4**, **L11**, **L12**), and no access modifier (**L3**, **L5**, **L9**, **L10**).
2. Uniqueness :*static* (all attributes and methods) and non-static (not in a utility class)
3. Assignable :*final* (**L2**, **L4**) means it is a constant value and can never be assigned, and non-final attributes are variables.

Structure of Utility Classes: Example (1.2)

```
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Each utility class contains a list of attributes and methods:

1. **L2 – L4**: Three attributes `RADIUS_TO_DIAMETER`, `radius`, `PI`
   - Each of these attributes has an initial value (2, 10, and 3).
   - Only the value of `radius` (non-final) may be changed.
2. **L5 – L13**: Six methods:
   - 1 **Mutator** (with the return type `void`): `setRadius(int newRadius)`
   - 5 **Accessors** (with an explicit `return` statement):

     e.g., `getDiameter()`, `getCircumference(int radius)`

Structure of Utility Classes: Example (1.3)

```
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

Each method has a (possibly empty) list of *parameters* (i.e., inputs) and their types:

- e.g., `getDiameter` (**L5**) has no parameters (i.e., it takes no inputs for its computation)
- e.g., `setRadius` (**L10**) has one parameter (i.e., `newRadius` of type `int`)

We talk about *parameters* in the context of method declarations.

Structure of Utility Classes: Example (1.4)

```
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

When the name of a method parameter clashes with the name of an attribute (**L9**):

- Any mention about that name (e.g., `radius`) refers to the parameter, not the attribute anymore.
- To refer to the attribute, write: `Utilities.radius`
- If you know what you're doing, that's fine; otherwise, use a different name (e.g., **L10**) to avoid unintended errors.

Structure of Utility Classes: Example (1.5)

```
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

The body (i.e., what's written between { and }) of a method (accessor or mutator) may:

1. Declare local variables (e.g., **L6**) to store intermediate computation results.

   The scope of these local variables is only within that method.
2. Perform assignments to change values of either local variables (**L6**) or attributes (**L10**).

Structure of Utility Classes: Example (1.6)

```
1  public class CircleUtilities {
2    private static final int RADIUS_TO_DIAMETER = 2;
3    static int radius = 10;
4    public static final int PI = 3;
5    static int getDiameter() {
6      int diameter = radius * RADIUS_TO_DIAMETER;
7      return diameter;
8    }
9    static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
10   static void setRadius(int newRadius) { radius = newRadius; }
11   public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
12   public static int getCircumference1() { return getDiameter() * PI; }
13   private static int getCircumference2() { return getCircumference(radius); }
14 }
```

A method body may *call* another method (i.e., *reuse* code):

3. Call a utility accessor and use (e.g., store, print, return) its return value: **L11** and **L13**.

- **L11**: Since we are in the same class, we do not need to write `CircleUtilities.getDiameter(radius)`
- **L11**: `getDiameter(radius)` passes method *parameter* `radius` as an *argument* value to method `getDiameter(...)`
- **L11**: It is equivalent to write (without reusing any code):
  `return radius * RADIUS_TO_DIAMETER * PI`

Structure of Utility Classes: Exercise

```
public class CircleUtilities {
  private static final int RADIUS_TO_DIAMETER = 2;
  static int radius = 10;
  public static final int PI = 3;
  static int getDiameter() {
    int diameter = radius * RADIUS_TO_DIAMETER;
    return diameter;
  }
  static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
  static void setRadius(int newRadius) { radius = newRadius; }
  public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
  public static int getCircumference1() { return getDiameter() * PI; }
  private static int getCircumference2() { return getCircumference(radius); }
}
```

Is the body of method `getCircumference1` equivalent to the body of method `getCircumference2`? Why or why not?

Structure of Utility Classes: Example (1.7)

```java
public class CircleUtilities {
  private static final int RADIUS_TO_DIAMETER = 2;
  static int radius = 10;
  public static final int PI = 3;
  static int getDiameter() {
    int diameter = radius * RADIUS_TO_DIAMETER;
    return diameter;
  }
  static int getDiameter(int radius) { return radius * RADIUS_TO_DIAMETER; }
  static void setRadius(int newRadius) { radius = newRadius; }
  public static int getCircumference(int radius) { return getDiameter(radius) * PI; }
  public static int getCircumference1() { return getDiameter() * PI; }
  private static int getCircumference2() { return getCircumference(radius); }
}
```

A method body may *call* another method (i.e., *reuse* code):

4. Call a utility mutator to change some data.

We will see an example about this later.

Visualizing a Utility Class

All *static* attributes and methods collectively make their resident utility class a **single** (i.e., one that cannot be duplicated) machine, which contains:

- Current values of attributes
- Definitions of methods (i.e., how computation is to be executed)

CircleUtilities	
RADIUS_TO_DIAMETER	2
radius	10
PI	3

getDiameter()	int diameter = radius * RADIUS_TO_DIAMETER;
	return diameter;

setRadius(int newRadius)	radius = newRadius;
getCircumference(int radius)	return getDiameter(radius) * PI;
getCircumference1()	return getDiameter() * PI;
getCircumference2()	return getCircumference(radius);

Using a Utility Class (1)

- We can either access a static attribute or call a static method in a utility class using its name.
- e.g., the method call `CircleUtilities.setRadius(40)` passes the value `40` as *argument*, which is used to instantiate every occurrence of the method *parameter* `newRadius` in method `setRadius` by `40`.

```
void setRadius(int ~~newRadius~~ 40) {
  radius = ~~newRadius~~ 40;
}
```

- Consequently, the effect of this method call is to change the current value of `CircleUtilities.radius` to `40`.

Entry Point of Execution: the “main” Method

The *main* method is treated by Java as the *starting point* of executing your program.

```
public class CircleUtilitiesApplication {
  public static void main(String[] args) {
    /* Your programming solution is defined here. */
  }
}
```

The execution starts with the first line in the *main* method, proceed line by line, from top to bottom, until there are no more lines to execute, then it *terminates*.

Using a Utility Class (2.1)

```java
public class CircileUtilitesApplication {
 public static void main(String[] args) {
   System.out.println("Initial radius of CU: " + CircleUtilities.radius);
   int d1 = CircleUtilities.getDiameter();
   System.out.println("d1 is: " + d1);
   System.out.println("c1 is: " + CircleUtilities.getCircumference1());
   System.out.println("======");
   System.out.println("d2 is: " + CircleUtilities.getDiameter(20));
   System.out.println("c2 is: " + CircleUtilities.getCircumference(20));
   System.out.println("======");
   System.out.println("Change the radius of CU to 30...");
   CircleUtilities.setRadius(30);
   System.out.println("======");
   d1 = CircleUtilities.getDiameter();
   System.out.println("d1 is: " + d1);
   System.out.println("c1 is: " + CircleUtilities.getCircumference1());
   System.out.println("======");
   System.out.println("d2 is: " + CircleUtilities.getDiameter(20));
   System.out.println("c2 is: " + CircleUtilities.getCircumference(20));
}
}
```

Executing it, what will be output to the console?

Using a Utility Class (2.2)

```
Initial radius of CU: 10
d1 is: 20
c1 is: 60
======
d2 is: 40
c2 is: 120
======
Change the radius of CU to 30...
======
d1 is: 60
c1 is: 180
======
d2 is: 40
c2 is: 120
```

Using a Utility Class: Client vs. Supplier (1)

- A *supplier* implements/provides a service (e.g., microwave).
- A *client* uses a service provided by some supplier.
  - The client must follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
  - If instructions are followed, the client would **expect** that the service does what is required (e.g., a lunch box is heated).
  - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* os the two parties?

	benefits	*obligations*
CLIENT	obtain a service	follow instructions
SUPPLIER	give instructions	provide a service

- There is a *contract* between two parties, violated if:
  - The instructions are not followed. [Client's fault]
  - Instructions followed, but service not satisfactory. [Supplier's fault]

Using a Utility Class: Client vs. Supplier (2)

```
class CUtil {
  static int PI = 3;
  static int getArea(int r) {
    /* Assume: r positive */
    return r * r * 3;
  }
}
```

```
1 class CUtilApp {
2   public static void main(...) {
3     int radius = ???;
4     println(CUtil.getArea(radius));
5 } }
```

- Method call **`CircleUtilities.getArea(radius)`**, inside class **`CircleUtilitiesApp`**, suggests a ***client-supplier relation***.
  - **Client**: resident class of the static method call [`CUtilApp`]
  - **Supplier**: context class of the static method [`CUtil`]
- What if the value of ??? at **L3** of `CUtilApp` is `-10`?

```
300
```

- What's wrong with this?
  - Client `CUtil` mistakenly gives illegal circle with radius `-10`.
  - Supplier `CUtil` should have reported a *contract violation*!

Using a Utility Class: Client vs. Supplier (3)

- *Method Precondition* : supplier’s assumed circumstances, under which the client can expect a satisfactory service.
  - Precondition of `int divide(int x, int y)`? [`y != 0`]
  - Precondition of `int getArea(int r)`? [`r > 0`]
- When **supplier** is requested to provide service with *preconditions* ***not*** satisfied, *contract is violated* by **client**.
- *Precondition Violations* ≈ `IllegalArgumentException`.
  Use `if-elseif` statements to determine if a violation occurs.

```
class CUtil {
 static int PI = 3;
 static int getArea(int r) throws IllegalArgumentException {
  if(r < 0) {
   throw new IllegalArgumentException("Circle radius " + r + "is not positive.");
  }
  else {
   return r * r * PI;
  }
 }
}
```

Documenting Your Class using Javadoc (1)

There are three types of comments in Java:

- `//` [line comment]
- `/* */` [block comment]
  - These two types of comments are only for you as a **supplier** to document interworking of your code.
  - They are hidden from **clients** of your software.
- `/** */` [block documentation]
  - This type of comments is for **clients** to learn about how to use of your software.

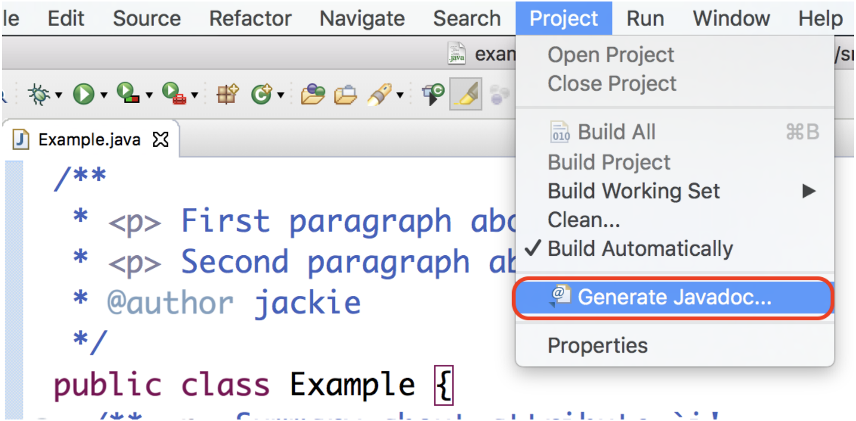
Documenting Classes using Javadoc (2.1)

```
/**
 * <p> First paragraph about this class.
 * <p> Second paragraph about this class.
 * @author jackie
 */
public class Example {
  /** <p> Summary about attribute ‘i’
    * <p> More details about ‘i’
    */
  public static int i;
  /**
    * <p> Summary about accesor method ‘am’ with two parameters.
    * <p> More details about ‘am’.
    * @return Always false for some reason.
    * @param s Documentation about the first parameter
    * @param d Documentation about the second parameter
    */
  public static boolean am (String s, double d) { return false; }
  /**
    * <p> Summary about mutator method ‘mm’ with no parameters.
    * <p> More details about ‘mm’.
    */
  public static void mm () { /* code omitted */ }
}
```

- Use ***@return*** only for mutator methods (i.e., returning non-`void`).
- Use ***@param*** for each input parameter.

Documenting Classes using Javadoc (2.2)

Generate an HTML documentation using the Javadoc tool supported by Eclipse:



```
/**
 * <p> First paragraph abo
 * <p> Second paragraph ab
 * @author jackie
 */
public class Example {
```

Exercises

- Implement a utility class named `Counter`, where
  - There is a static integer counter `i` whose initial value is `5`.
  - There is a static constant maximum `MAX` of value `10` for counter `i`.
  - There is a static constant minimum `MIN` of value `10` for counter `i`.
  - Your implementation should be such that the counter value can never fall out of the range $[5, 10]$.
  - There is a mutator method `incrementBy` which takes an integer input parameter `j`, and increments the counter `i` value by `j` if possible (i.e., it would not go above `MAX`).
  - There is a mutator method `decrementBy` which takes an integer input parameter `j`, and decrements the counter `i` value by `j` if possible (i.e., it would not go below `MIN`).
  - There is an accessor method `isPositive` which takes an integer input parameter `j`, and returns `true` if `j` is positive, or returns `false` if otherwise.
- Properly document your `Counter` class using Javadoc and generate the HTML documentation using Eclipse.

Index (1)

Index (2)

Index (3)