CSE 331 18au Section 1 – Specifications

NOTE* - Some problems in this worksheet mistakenly dealt with null ints, which are impossible because ints are primitive and primitives cannot be null. These issues have been fixed in this version of the worksheet.

1. *Alice must write a method* `histogram` *that takes in an array of integers* `sleepData` *that corresponds to answers from a survey about how many hours college students sleep, with possible answers ranging from* `0` *to* `9`. `histogram` *then returns an array of integers of size* `10`*, where the value at position* `i` *is the number of times* `i` *appeared in* `sleepData`*. For example, if* `sleepData = [3,4,6,7,2,1,4]`*, then* `histogram` *returns* `[0,1,1,1,2,0,1,1,0,0]`*. If* `sleepData` *is* `null`*, throw a* `NullPointerException`*, and if* `sleepData` *is empty, return* `null`*. Fill out* `histogram's` *specification*:

   ```
   /**
   * Takes in an array sleepData of hours slept by college students, and converts it into a
   *histogram
   * @spec.requires 0 <= sleepData[i] <= 9 for all possible i
   *
   * @spec.modifies _____
   *
   * @spec.effects _____
   *
   * @return A histogram of the hours slept, in the format of a 10 element array of ints
   *where pos i of the array contains the number of times a student reported sleeping for i
   *hours. Returns null if sleepData is empty
   *
   * @throws NullPointerException if sleepData is null
   */
   public int[] histogram (int[] sleepData) {
   ```

2. *Implement the following specification:*

   ```
   /** Takes in the longest side of a triangle, longestSide, and a second side of the
   triangle, side, and returns the shortest possible length of the last side
   * @spec.requires longestSide >= leg
   * @returns the smallest possible integer value of the last side of the triangle.
   * @throws IllegalArgumentException if longestSide <= 0 or
   ```

```
*side <= 0
**/

Public int shortestSideLength (int longestSide, int side) {
   if (longestSide <= 0 || side <= 0) {
      throw new IllegalArgumentException();
   }
   return longestSide – side + 1;
}
```

3. *Suppose we have a* `BankAccount` *class with instance variable* `balance`. *Consider the following three specifications for a* `BankAccount` *method* `withdraw`, *which takes in an int* `amount` *that signifies the amount the user wants withdrawn from the balance:*

   A. **@spec.modifies** `balance`
   @spec.effects *decreases* `balance` *by* `amount`.

   B. **@spec.modifies** `balance`
   @spec.requires `amount >= 0` ***and*** `amount <= balance`
   @spec.effects *decreases* `balance` *by* `amount`.

   C. **@spec.modifies** `balance`
   @spec.effects *decreases* `balance` *by* `amount`
   @throws `InsufficientFundsException` *if* `balance < amount`

   Which specifications do each of these implementations meet? Write **A, B,** *and/or* **C** *for each implementation.*

   I.
   ```
   void withdraw(int amount) {
        balance -= amount;
   }
   ```
   Specifications: _____*A,B*_____

   II.
   ```
   void withdraw(int amount) {
        if (balance >= amount) {
             balance -= amount;
        }
   }
   ```
   Specifications: _____*B*_____

   III.
   ```
   void withdraw(int amount) {
   ```

```
        if (amount < 0) {
            throw new IllegalArgumentException();
        }
        balance -= amount;
    }
```

Specifications: _____B_____

IV.
```
    void withdraw(int amount) throws
    InsufficientFundsException {
        if (balance < amount) {
            throw new InsufficientFundsException();
        }
        balance -= amount;
    }
```

Specifications: _____B,C_____

4. (Midterm 15wi Problem 4) *Here is the header for a method that computes a student's overall score and adds that information to a gradebook data structure:*

```
void addScore(String name, List scores, Map gradeBook);
```

A. *Here are two possible specifications for this method:*

X

@spec.requires `name != null` *and* `scores != null` *and* `gradeBook != null`
@spec.modifies `gradebook`
@spec.effects add a mapping to `gradebook`

Y

@spec.requires `name != null` *and* `scores != null`
@spec.modifies `gradebook`
@spec.effects add a mapping to `gradebook`
@throws `IllegalArgumentException` *if* `gradeBook` *is* `null`

Which specification is stronger than the other? (circle) X <u>Y</u> neither

B. *Here is one possible implementation of this method:*

```
if (name == null || scores == null || gradeBook == null)
{
  throw new IllegalArgumentException();
}
double grade = 0.0;
for (double s : scores) {
```

```
    grade += s;
  }
  if (scores.size() > 0) {
    grade /= scores.size();
  }
  gradeBook.put(name,grade);
```

Which specification(s) does this implementation satisfy? (circle) X Y both neither

5. *(Midterm 17AU Problem 1) Alice is writing a function* `bestDeal` *that takes in an array* `prices` *and then returns the smallest price. She intends to implement* `bestDeal` *by sorting* `prices`*, but she does not want clients to depend on* `prices` *being sorted.*

   A. *Write a specification for her function:*

```
/**
*@spec.requires prices != null
*@spec.modifies prices
*@returns lowest price in prices
*
*
*/
int bestDeal(int[] prices) { …
```

   B. *Suppose that Alice decides to change her implementation to no longer sort* `prices`*. How should she change the specification above?*

   Remove the @modifies clause

   C. *This new specification would be (circle one):* weaker incomparable stronger

   D. *Suppose that Alice decides instead to stick with the version that sorts* `prices` *but will now allow clients to depend on that behavior. How should she change the specification above?*

Add an @effects clause that says that prices is sorted

E. *This new specification would be (circle one):* weaker incomparable stronger