Quiz Section 1: Specifications – Solutions

Task 1 – At Your Own Pair-il

Suppose `price` is an integer and `items` is an array. For each pair of assertions, compare them and state which one is stronger and which one is weaker (reminder that a stronger assertion implies a weaker one meaning that if the stronger one is true, the weaker one must be true as well!), or if they are incomparable:

a) "`price > 0 and items is not null`" or "`items is not null`"

"`price > 0 and items is not null`" is the stronger assertion

b) "`price > 0`" or "`price >= 0`"

"`price > 0`" is the stronger assertion because if `price > 0`, we know that `price >= 0` must be true as well.

c) "`items is an array`" or "`items is a sorted array`"

"`items is a sorted array`" is the stronger assertion

d) "`price > 0`" or "`items is not null`"

These two are incomparable

Task 2 – The Same Ol' Strong and Dance

We plan to provide the following method:

```
/** Returns the best menu item for the given price.
 * ...
 */
public static MenuItem findBest(int price, MenuItem[] items);
```

To do so, we need to fill in the rest of the specification. For example, we need to explain exactly which item will be returned. The term "best" is far too vague.

We are considering the following alternatives:

```
@requires price > 0 and items is not null                          // Spec A
@return an item T in items with T.price <= price <= T.price + 0.5
    or null if none exists

@requires items is not null                                        // Spec B
@throws IllegalArgumentException if price <= 0
@return an item T in items with T.price <= price <= T.price + 0.5
    or null if none exists

@requires price > 0 and items is not null                          // Spec C
@return the item in items whose price is closest to the given price
    but not more than the given price or null if none exists

@requires price > 0                                                // Spec D
@throws NullPointerException if items is null
@return the item in items whose price is closest to the given price
    but not more than the given price or null if none exists

@requires price > 0                                                // Spec E
@return an item T in items with T.price <= price <= T.price + 0.5
    or null if none exists or if items is null
```

a) Fill in the following table explaining the relationships between each pair of specifications. Write an "S" if the spec on left (the row) is stronger than the spec on top (the column), a "W" if the left spec is weaker, and "—" if the specs are incomparable.

	A	**B**	**C**	**D**	**E**
A	X				
B		X			
C			X		
D				X	
E					X

	A	**B**	**C**	**D**	**E**
A	X	W	—	—	W
B	S	X	—	—	—
C	—	—	X	W	—
D	—	—	S	X	—
E	S	—	—	—	X

- A and C/D are incomparable
- A is weaker than B as A has more restrictive inputs (price > 0)
- A is also weaker than E as A has more restrictive inputs (items is not null) and less restrictive outputs (null if items is null).
- C is weaker than D as C has more restrictive inputs (items is not null). While D throws an exception on that same input, it still "accepts" the input and actually has more guarantees (more restrictive output) as a result.

b) Now consider a new pair of precondition and postcondition behaviors:

```
@requires price > 0
@return an item T in items with T.price <= price <= T.price + 0.5
```

This specification is not sensible, what is wrong with it? Why shouldn't we use it?

The specification does not describe what should happen if/when items is null. The client could make a guess (maybe it treats it like an empty array?), but the specification simply doesn't say. This is a problem because a client cannot effectively use a method if they do not know what happens in some cases.

Task 3 – Great Finds Think Alike

For each of the following implementations, state which of the specifications it satisfies. If it does not satisfy some specification, explain (in as few words as possible) why it does not.

a)
```
public static MenuItem findBest(int price, MenuItem[] items) {
  if (price <= 0)
    throw new IllegalArgumentException("bad price");
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price && price <= items[i].price + 0.5)
      return items[i];
  }
  return null;
}
```

This satisfies A and B. It does not satisfy C or D because the item returned is not necessarily the one nearest below in price. It does not satisfy E because it does not return null when items is null.

b)
```
public static MenuItem findBest(int price, MenuItem[] items) {
  items.sort();  // puts items in order by increasing price
  MenuItem best = null;
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price)
      best = items[i];
  }
  return best;
}
```

This satisfies C and D. Note that the code does in fact throw a `NullPointerException` when `items` is null, even though there is no explicit check for that case. This does not satisfy A, B, or E because (amongst other reasons) the return value is not necessarily within the required price range.

c)
```
public static MenuItem findBest(int price, MenuItem[] items) {
  if (items == null)
    return null;
  if (price <= 0)
    throw new IllegalArgumentException("bad price");
  for (int i = 0; i < items.length; i++) {
    if (items[i].price <= price && price <= items[i].price + 0.5)
      return items[i];
  }
  return null;
}
```

This satisfies A, B, and E. It does not satisfy C or D because the item returned is not necessarily the one nearest below in price.