Designing Contracts

This week, we are switching gears by studying techniques that code developers can employ to ensure the correctness and robustness of the code they write.

1. Wait for the instructor to demo pre- and post- conditions with `calculateTax`. Now is your turn - come up with pre- and post- conditions for `add` and `remove` methods in `Basket` class:

```
public class Basket {
  private BigDecimal totalValue = BigDecimal.ZERO;
  private Map<Product, Integer> basket = new HashMap<>();
  public void add(Product product, int qtyToAdd) {
    assert product != null : "Product is required";
    assert qtyToAdd > 0 : "Quantity has to be greater than zero";

    // add the product
    // update the total value

    assert basket.containsKey(product) :
                "Product was not inserted in the basket";
    assert totalValue.compareTo(oldTotalValue) == 1 :
             "Total value should be greater than previous total value";
    assert totalValue.compareTo(BigDecimal.ZERO) >= 0 :
                "Total value can't be negative.";

  }
  public void remove(Product product) {
    assert product != null :
                "product can't be null";
    assert basket.containsKey(product) :
                "Product must already be in the basket";

    // remove the product from the basket
    // update the total value

    assert !basket.containsKey(product) :
                "Product is still in the basket";
    assert totalValue.compareTo(BigDecimal.ZERO) >= 0 :
                "Total value can't be negative.";

  }
}
```

Consider the following code:

```
class Board{
    Square[][] board;

    // constructors and other methods

    public Square squareAt(int x, int y){
       assert x >= 0;
       assert x < board.length;
       assert y >= 0;
       assert y < board[x].length;
       assert board != null;
       Square result = board[x][y];
       assert result != null;
       return result;
    }

    // other methods
}
```

2. Out of all the assertions which one or which combination of them could become the class invariant for the class `Board`?

```
assert board != null;
```

3. Suppose we remove the last assertion (`assertresult!=null`), which states that the result can never be `null`. Are the existing pre-conditions of the `squareAt` method enough to ensure the property of the removed assertion?

The existing pre-conditions are not enough to ensure the property of the removed assertion. The post-condition ensured that the returned value was never null. The pre-conditions ensure that board itself cannot be null and that x and y are in its range. But it does not ensure that values in board are not null.

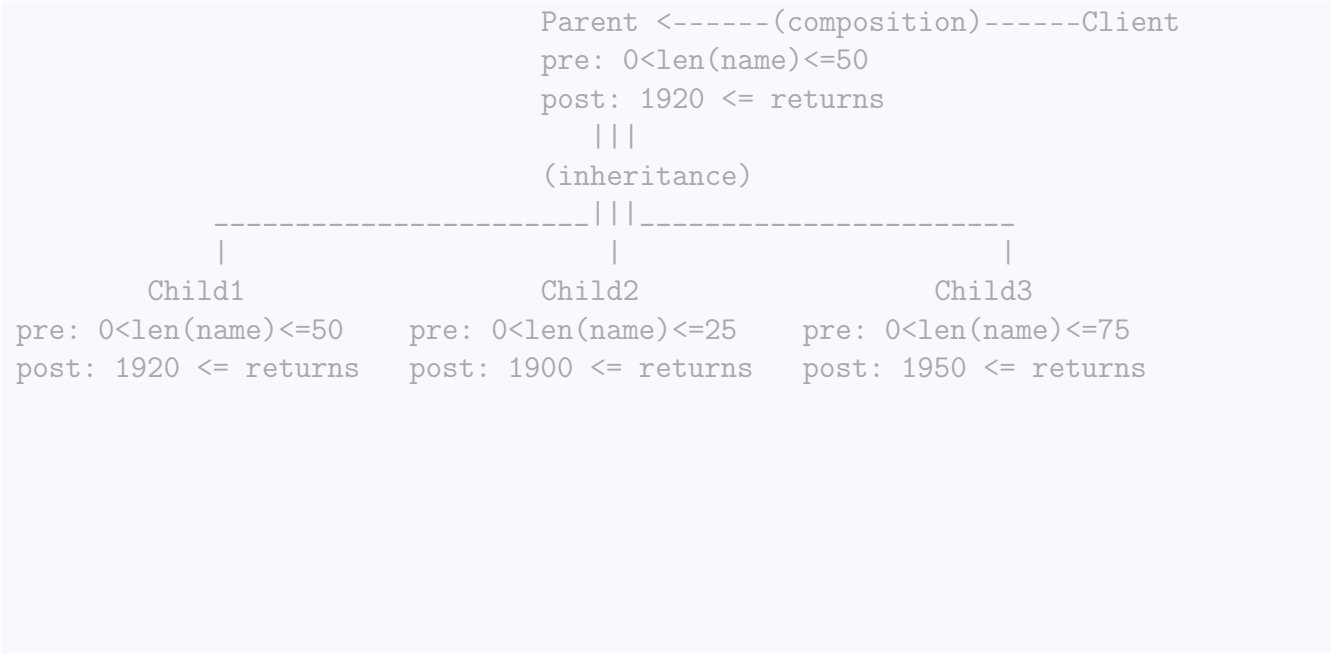
4. What can we add to the class (other than the just-removed post-condition) to guarantee this property?

To have the same guarantee, the class would need an invariant that ensures that no place in board is null.

Consider the following scenario of inheritance for the method `getBirthYear(String name)` that is implemented in the parent and then overloaded by each of the children classes. The pre- and post-conditions for the parent and children classes are as follows:

Parent Class	Child 1 Class	Child 2 Class	Child 3 Class
`len(name) upto 50`	`len(name) upto 50`	`len(name) upto 25`	`len(name) upto 75`
`returns >= 1920`	`returns >= 1920`	`returns >= 1900`	`returns >= 1950`

5. Draw the inheritance diagram assuming that the `Client` class uses a list of `Parent` objects.



6. Explain which overloaded method would fail and why (i.e., it could be due to the pre-condition or the post-condition or both)? Explain why.

Child 1 class won't fail because the pre and post- conditions are the same
Child 2 class will fail because the pre-condition is less inclusive and the post-condition is more inclusive
Child 3 class won't fail because the pre-condition is more inclusive and the post-condition is less inclusive

Consider the following documentation for `indexOf` method:

```
/**
 * Finds the index of the given value in the array starting at the
 * given index.
 *
 *
 * @param array
 *            the array to search through for the object, may be null
 * @param valueToFind
 *            the value to find
 * @param startIndex
 *            the index to start searching at
 * @return the index of the value within the array, INDEX_NOT_FOUND
 *         (-1) if not found or {@code null} array input
 */
public static int indexOf(final int[] array, final int valueToFind,
                                                int startIndex) {
    ...
}
```

7. What are the pre-conditions for this method?

Input array not null.
startIndex greater than 0.
startIndex less than the array length.

8. What are some stronger and softer ways of dealing with the invalid cases for this method in our code?

stronger
- throw exceptions
softer
- return INDEX_NOT_FOUND (-1) for null array and startIndex greater or equal than the array length.
- negative index ignored, start at 0.