

Solutions to Quiz 1 (March 14, 2016)

Problem 1 (Multiple Choice) (20 points).

(a) Which of the following must be true of an underdetermined function specification? (choose all that apply)

- A. An underdetermined spec means the implementation is unwritten.
- B. An underdetermined spec means the implementation is nondeterministic.
- C. An underdetermined spec allows multiple valid outputs for some input.
- D. An underdetermined spec allows multiple valid inputs that give some output.

Solution. C. None of the other options *must* be true. ■

(b) After the following code is executed, what is the value of array `arr`? (choose one answer)

```
final String[] arr = new String[2];
String s = "6.005";
arr[0] = s;
s = "is";
arr[1] = s;
String t = arr[0];
t = "fun";
```

- A. ["is", "is"]
- B. ["6.005", "is"]
- C. ["fun", "is"]
- D. *none of the above*

Solution. B. The reassignment of `s` to "is" or `t` to "fun" do not affect `arr[0]`. ■

(c) The line of Java code `String t = arr[0];` involves... (choose all that apply)

- A. assignment
- B. equivalence relation
- C. mutation
- D. static typing

Solution. A, D. This line *assigns* to `t`, which has *static type* `String`. ■

(d) Alyssa P. Hacker is designing an immutable type to represent users in her computer system. The User's login name is stored as:

```
private final String kerberos;
```

She defines two `User` objects as equal if their login names are the same, **ignoring case**:

```
@Override public boolean equals(Object other) {
    if (!(other instanceof User)) { return false; }
    User that = (User)other;
    return this.kerberos.equalsIgnoreCase(that.kerberos);
}

@Override public int hashCode() { /* TODO */ }
```

Which of the following implementations of `hashCode()` would be valid, satisfying the `Object` contract? (choose all that apply)

- A. `return 31;`
- B. `return this.kerberos.hashCode();`
- C. `return this.kerberos.toLowerCase().hashCode();`
- D. `return this.kerberos.toUpperCase().hashCode();`

Solution. A, C, D.

A is correct, although not necessarily a good hash function.

C and D are correct because `Strings` that are equal ignoring case will have the same value for both `toLowerCase()` and `toUpperCase()`.

B is incorrect because two `Users` with login names that differ only in case will have different hash codes even though the `User` objects are equal. ■

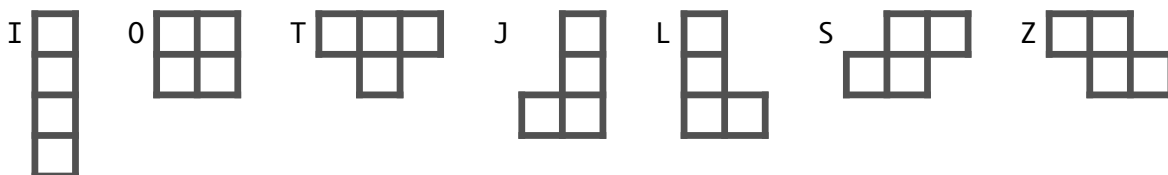
(e) If the code in answer choice (A) above appeared in your 6.005 code review assignment, which of the following comments would be appropriate criticisms? (choose all that apply)

- A. The code isn't DRY.
- B. The code uses magic numbers.
- C. The code exposes `User`'s representation.
- D. The code is unnecessary, we don't need to override `hashCode` if we only return a constant.

Solution. B. 31 is a magic number. ■

Problem 2 (Specifications) (20 points).

A **tetromino** is a shape made out of four adjacent squares. These shapes are most famous from the game *Tetris* where the player must rotate and translate falling tetrominoes in order to fit them together. There are seven possible tetrominoes that lie on the 2D plane, and each is identified by a letter it looks like:



Define a **tetromino shape letter** as one of the seven letters I O T J L S Z, either upper- or lowercase. Tetrominoes may be rotated by 0, 90, 180, or 270 degrees.

A diagram of all seven tetrominoes in all four orientations is on the last page of this quiz.

Let's define an abstract data type to represent tetrominoes:

```
public class Tetromino {

    // determine if a character is a valid tetromino shape letter
    public static boolean isValidShape(char shape) { ... }

    // make a new tetromino
    public Tetromino(char shapeLetter, int rotationDegrees) { ... }

    // rotate this tetromino
    public void rotateClockwise() { ... }

    // get the shape of this tetromino
    public char shape() { ... }
}
```

(a) Fill in this table with information about the operations of Tetromino.

The answers for `isValidShape` are already given.

operation	type signature	classify the type of ADT operation	Java implementation strategy
<code>isValidShape</code>	<code>char → boolean</code>	not applicable	static method
<code>Tetromino</code>	<code>char × int → Tetromino</code>	creator	constructor
<code>rotateClockwise</code>	<code>Tetromino → void</code>	mutator	instance method
<code>shape</code>	<code>Tetromino → char</code>	observer	instance method

(b) Consider these different specifications for the `isValidShape` function:

```
public static boolean isValidShape(char shape)

    /**
Spec A   * @param shape any character
         * @return true iff shape is a lowercase tetromino shape letter
         */

    /**
Spec B   * @param shape an English alphabetic character
         * @return true iff shape is a lowercase tetromino shape letter
         */
```

```

    /**
Spec C   * @param shape a lowercase tetromino shape letter
         * @return true
         */

```

Compare these specifications. For each pair below, **circle one correct option** and write **write a brief explanation** to complete the sentence.

A vs. B:

```

Spec A is  equivalent to
           weaker than
           stronger than
           incomparable to
           spec B because

```

Solution. Spec A is stronger than spec B because its precondition is weaker and its postcondition is the same. ■

B vs. C:

```

Spec B is  equivalent to
           weaker than
           stronger than
           incomparable to
           spec C because

```

Solution. Spec B is stronger than spec C because it has a weaker precondition, and it satisfies C's postcondition for any input that satisfies C's precondition. ■

Problem 3 (AFs & RIs) (20 points).

Here is an implementation of the Tetromino ADT.

```

/** A rotatable tetromino. */
public class Tetromino {

    // ... static method isValidShape ...

    private final char shape;
    private int rotation;

    // Abstraction function
    // TODO
    // Representation invariant
    // TODO
    // Safety from rep exposure
    // TODO

    /**
     * Make a new Tetromino with the given shape and rotation.
     * @param shapeLetter uppercase tetromino shape letter
     * @param rotationDegrees clockwise rotation in degrees,
     *                       must be 0, 90, 180, or 270
     */
    public Tetromino(char shapeLetter, int rotationDegrees) {

```

```

        this.shape = shapeLetter;
        this.rotation = rotationDegrees / 90;
    }

    /**
     * TODO
     */
    public void rotateClockwise() {
        rotation = (rotation + 1) % 4;
    }

    /**
     * @return shape of this tetromino: 'I' 'O' 'T' 'J' 'L' 'S' or 'Z'
     */
    public char shape() {
        return shape;
    }

    @Override
    public String toString() {
        return shape + "-shape@" + rotation * 90 + "deg";
    }
}

```

(a) For each of the statements below, say whether it should be included in the internal documentation of Tetromino by writing:

AF if the statement belongs in the abstraction function

RI ... the rep invariant

EXP ... the argument that type has no rep exposure

NONE if it should not be included in any of those

You should include in the AF, RI, or EXP **all good statements** that are compatible with the code and specs on the previous page.

Do not include statements that are not compatible with the code and specs.

shape is private and an immutable value

rotation is private and an immutable value

this Tetromino is never returned to clients

shape is an uppercase tetromino shape letter

shapeLetter is an uppercase tetromino shape letter

the tetromino has the shape given by shape

rotation = 0

the tetromino is rotated clockwise by rotation

the tetromino is rotated clockwise rotation times 90 degrees

0 <= rotation < 4

0 <= rotation < 360

rotation is one of { 0, 90, 180, 270 }

Solution.

EXP

EXP

NONE (doesn't make sense)

RI

NONE (part of the precondition of the constructor)

AF

NONE

NONE

AF (see e.g. toString)

RI (established by constructor, maintained by rotateClockwise)

NONE

NONE

■

Consider these different specifications for the rotateClockwise method:

```
public void rotateClockwise()
```

For each possible specification below, **write a one-sentence code review comment that identifies the most serious problem** with the spec.

(b) /**
 * *Update this tetromino's rotation number to add 1, mod 4.*
 */

Comment:

Solution. The specification must be abstract, it should not talk about the representation.

The specification should be declarative, not operational. ■

(c) /**
 * *@return this tetromino rotated by 90 degrees clockwise*
 */

Comment:

Solution. The specification, which says a Tetromino is returned, does not agree with return type void. ■

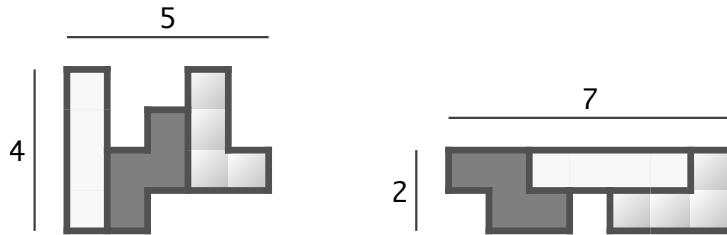
(d) /**
 * *Rotate this tetromino (does not otherwise modify the tetromino).*
 */

Comment:

Solution. This specification is too weak, it does not specify the amount of rotation. ■

Problem 4 (Testing) (20 points).

Let's consider the problem of packing tetrominoes into a rectangle by rotating and translating them in the 2D plane. For example, here are tetrominoes I, L, & Z packed into a 5×4 rectangle, and a 7×2 rectangle:



For those three tetrominoes, the minimum area they can be packed into is $7 \times 2 = 14$ squares.

(a) /**

```
* @param shapes string of tetromino shape letters
* @return a 2-element list representing a minimum-area rectangle into which
*         the tetrominoes given by shapes can be packed
*         (for example, pack("ILZ") might return the list [ 2, 7 ])
*/
```

```
public static List<Integer> pack(String shapes) { ... }
```

Start writing a black box testing strategy for `pack(...)` by giving **one good partitioning** for input shapes:

For each of the test cases below, in the first box **write YES or NO** in the first box to say whether the test is valid or not. If the test is not valid, **write a one-sentence reason why not**. We'll use Python's syntax to represent lists for brevity.

(b) `shapes = "X"`
`rectangle = [0, 0]`

Valid? Reason if invalid:

Solution. Invalid: shape X violates the precondition. ■

(c) `shapes = "I"`
`rectangle = [1, 4]`

Valid? Reason if invalid:

Solution. Invalid: rectangle [4, 1] is also allowed by the postcondition. ■

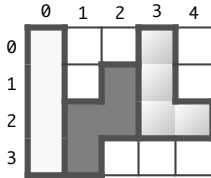
(d) `shapes = "L0"`
`rectangle = [3, 3]`

Valid? Reason if invalid:

Solution. Valid. ■

Problem 5 (Rep Exposure) (20 points).

Let's define a mutable abstract data type TetrominoGrid to represent tetrominoes arranged on a fixed-size grid, where every tetromino fits on the grid and none of the tetrominoes overlap. For example:

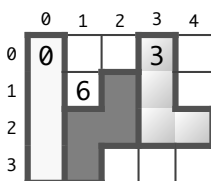


On the next page is an implementation of TetrominoGrid. In its rep, TetrominoGrid stores the location of each tetromino using a map from integers to tetrominoes. The integer keys are square numbers:

Each square in the grid is numbered starting from 0 in the upper-left corner.

On a grid of width *width*, square at row *row* and column *col* is numbered $row \times width + col$.

For example, here's how our type would represent the example above:



width = 5, height = 4

tetrominoLocations = {

0: (Tetromino: I-shape rotated 0 degrees),

3: (Tetromino: L-shape rotated 0 degrees),

6: (Tetromino: Z-shape rotated 90 degrees)

}

The TetrominoGrid ADT is implemented using the same **mutable** Tetromino ADT from previous questions.

(a) Identify all instances of rep exposure in TetrominoGrid on the next page. For each one, write:

1. the line number most directly responsible for the problem,
2. at most one-sentence description of the rep exposure, and
3. at most one-sentence description of how to fix the problem, or a single corrected line of code.

There may be more boxes than you need.

Line #: Explanation:

Line #: Explanation:

Line #: Explanation:

Line #: Explanation:

Line #: Explanation:

Solution.

Line 3: tetrominoLocations can be accessed and mutated by clients. It should be private.

Line 8: returns a Tetromino that is in the rep, clients could mutate it and break the rep invariant. Should return a defensive copy.

Line 11 (or 10): returns an unmodifiable list, but the Tetromino objects are still mutable parts of the rep. Should return defensive copies. ■

You may detach this page from your quiz, but you must write your name above and turn in all pages.

/**

* Mutable type representing a fixed-size grid with a valid arrangement of
 * tetrominoes: every tetromino fits on the grid without overlapping.

*/


```
public class TetrominoGrid {  
  
1) private final int width;  
2) private final int height;  
3) public final Map<Integer, Tetromino> tetrominoLocations;  
  
    // create a new grid and try to add some tetrominoes  
    public TetrominoGrid(int width, int height, List<Tetromino> initial) {  
4)         this.width = width;  
5)         this.height = height;  
6)         this.tetrominoLocations = new HashMap<>();  
           for (Tetromino tetromino : initial) {  
7)             this.add(tetromino.shape());  
           }  
    }  
  
    // get the tetromino whose upper-left corner is the given square  
    public Tetromino getTetrimino(int row, int col) {  
8)         return tetrominoLocations.get(row * width + col);  
    }  
  
    // return a list of all tetriminos currently on the grid  
    public List<Tetromino> getTetriminosOnBoard() {  
9)         List<Tetromino> tetrominoes = new ArrayList<>();  
           for (Tetromino tetromino : tetrominoLocations.values()) {  
10)            tetrominoes.add(tetromino);  
           }  
11)        return Collections.unmodifiableList(tetrominoes);  
    }  
  
    // try to add a tetromino to the grid, if it can fit without overlap  
    public boolean add(char shape) {  
12)        boolean canBePlaced = false; // can we fit the tetromino anywhere?  
13)        int topLeft = -1;  
14)        int rotation = 0;  
           // ... code to check whether the new tetromino fits and doesn't overlap ...  
           // ... updates the values of the local variables accordingly ...  
           if (canBePlaced) {  
15)            tetrominoLocations.put(topLeft, new Tetromino(shape, rotation));  
           }  
16)        return canBePlaced;  
    }  
  
    // ... other operations ...  
}
```

You may detach this page from your quiz, but you must write your name above and turn in all pages.

	0 degrees	90 degrees	180 degrees	270 degrees
I				
O				
T				
J				
L				
S				
Z				