Reference Mystery [Reference Semantics Review]

```
import java.util.*;     // for Arrays classes

public class ReferenceMystery {
    public static void main(String[] args) {
        int x = 1;
        int[] a = new int[2];
        // line 1
        mystery(x, a);
        // line 2
        System.out.println(x + " " + Arrays.toString(a));
        x--;
        a[1] = a.length;
        // line 3
        mystery(x, a);
        // line 4
        System.out.println(x + " " + Arrays.toString(a));
    }

    public static void mystery(int x, int[] list) {
        list[x]++;
        x++;
        System.out.println(x + " " + Arrays.toString(list));
    }
}
```

Enter the output that is produced by each line.

Question 1

line 1 output

No response

Question 2

line 2 output

No response

Question 3

line 3 output

No response

Question 4

line 4 output

No response

Longest Sorted Sequence [ArrayList Review]

Write a method `longestSortedSequence` that returns the length of the longest sorted sequence within a list of integers. A sequence is considered to be sorted when the numbers are in an order which is *non-decreasing*, e.g. (0, 0, 1, 2).

For example, if a variable called `list` stores the following sequence of values:

```
[1, 3, 5, 2, 9, 7, -3, 0, 42, 308, 17]
```

then the call: `longestSortedSequence(list)` would return the value 4 because it is the length of the longest sorted sequence within this list (the sequence -3, 0, 42, 308).

If the list is empty, your method should return 0.

Notice that for a non-empty list the method will always return a value of at least 1 because any individual element constitutes a sorted sequence.

sumLeftHalf [2D Array Review]

Write a method `sumLeftHalf` that takes an `int[][]` as a parameter and returns the sum of all elements in the left half of the array. Do not include the middle column in our sum if the 2D array has an odd number of columns.

For example, if we were given the following array:

```
int[][] arr = {{3, 1, 4, 1},
               {5, 9, 2, 6},
               {5, 3, 5, 8}}
```

then the call `sumLeftHalf(arr)` should return the value 26.

In the case of this array:

```
int[][] arr2 = {{5, 2, 3},
                {1, 3, 6},
                {5, 8, 2}}
```

then the call `sumLeftHalf(arr2)` should return the value 11.

Stacks and Queues Cheatsheet

Stack

A helpful analogy for a stack is a deck of cards. In a deck of cards, we can put a card on top of the deck, and remove a card from the top of the deck. Similarly, with a stack, we can add elements to the front of the stack and remove elements from the front of the stack.

Making a stack

```
Stack<Type> stack = new Stack<>();
```

Stack Methods

```
stack.empty()
```

This method returns a boolean, true if the stack is empty, and false otherwise.

```
stack.push(Type item)
```

This method will add item on top of the stack.

```
stack.pop()
```

This method will remove the top element of the stack, and return it.

```
stack.peek()
```

This method returns the top element on the stack, but does not remove it from the stack.

```
stack.size()
```

This method will return the number of items in the stack.

Queue

A queue is a data structure we all interact with day to day. When you get in line somewhere, you are entering a queue, where you were the last person to enter the line, therefore you will be the last to leave (unless more join before you leave). Queues can only be seen from the front of them typically, so you can see who is at the front, but not the middle of the line.

Making a queue

```
Queue<Type> queue = new LinkedList<>();
```

Queue Methods

```
queue.isEmpty()
```

This method returns a boolean, true if the queue is empty, and false otherwise.

```
queue.add(Type item)
```

This method will add item to the back of the queue.

```
queue.remove()
```

This method will remove the first item from the queue and return it.

```
queue.peek()
```

This method will return the first item from the queue, but not remove it.

```
queue.size()
```

This method will return the number of items in the queue.

collapse [Stacks and Queues Review]

Write a method collapse that takes a Stack of integers as a parameter and that collapses it by replacing each successive pair of integers with the sum of the pair. For example, suppose a stack stores this sequence of values:

```
bottom (7, 2, 8, 9, 4, 13, 7, 1, 9, 10) top
```

Assume that stack values appear from bottom to top. In other words, 7 is on the bottom, with 2 on top of it, with 8 on top of it, and so on, with 10 at the top of the stack.

The first pair should be collapsed into 9 (7 + 2), the second pair should be collapsed into 17 (8 + 9), the third pair should be collapsed into 17 (4 + 13) and so on to yield:

```
bottom (9, 17, 17, 8, 19) top
```

As before, stack values appear from bottom to top (with 9 on the bottom of the stack, 17 on top of it, etc). If the stack stores an odd number of elements, the final element is not collapsed. For example, the sequence:

```
bottom (1, 2, 3, 4, 5) top
```

would collapse into:

```
bottom (3, 7, 5) top
```

with the 5 at the top of the stack unchanged.

You are to use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You also may not solve the problem recursively.

In writing your method, assume that you are using the Stack and Queue interfaces and the ArrayStack and LinkedQueue implementations discussed in lecture. As a result, values will be stored as Integer objects, not simple ints.

Your method should take a single parameter: the stack to collapse.

⚠ This problem is significantly more challenging than the previous ones! Try and think about your algorithm before you jump into writing code

Map and Set Cheatsheet

Set<E>

- `add(value)` adds the given value to the set. If the value is already in the set, nothing happens
- `contains(value)` returns true if the given value is found in this set
- `remove(value)` removes the given value from the set
- `clear()` removes all elements of the set
- `size()` returns the number of elements in set
- `isEmpty()` returns true if the set's size is 0
- `toString()` returns a string such as "[3, 42, -7, 15]"

Map<K, V>

- `put(key, value)` associates the given `key` with the given `value`
- `get(key)` returns the value associated with the given `key` , or returns `null` if no mapping exists for the given `key`
- `containsKey(key)` returns true if and only if the map contains a mapping for the given `key`
- `remove(key)` removes the mapping for the given `key`
- `size()` returns the number of mappings in the map.
- `isEmpty()` returns true if and only if there are no mappings in the map
- `keySet()` returns a set of all keys in the map
- `values()` returns a collection of all values in the map
- `equals(other)` returns true if and only if the `other` map contains the mappings

⭐ rarestAge [Map Review]

Write a method `rarestAge` that takes a map from names (Strings) to ages (integers), and returns the least frequently-occurring age. Assume the maps are not null and no key is null.

Suppose a `map` contains the following entries:

```
{Trien=22,
 Sara=25,
 Ajay=25,
 Audrey=20,
 Ben=20,
 Eric=20,
 Joe=25,
 Joshua=25,
 Nicole=22,
 Rohini=21}
```

1 person is age 21, 3 people are age 20, 2 people are age 22, and 4 people are age 25. `rarestAge` returns 21 because only one person is age 21.

If there is a tie (multiple rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair `Andrew=21` to the map above, there would now be a tie of 2 people of age 21 and 2 people of age 22. `rarestAge` would still return 21 (as 21 is the smaller of 21 and 22).

Length Count [Sets Review]

Problem Description

Write a method called `lengthCount` that takes a `Set<String>` as a parameter and returns a `Map` mapping how many times a certain word length occurs in the `Set`.

For example, if a variable called `words` contains the following set of strings:

```
[to, be, or, not, that, is, the, question]
```

then the call **`lengthCount(words)`** should return a map whose values are counts of a certain String length and whose keys are the String lengths:

```
{2=4, 3=2, 4=1, 8=1}
```

Notice that the key 2 maps to 4 because there are 4 Strings that have a length of 2 (to, be, or, is). Note the ordering of the elements of the resultant Map.

Step 1 – Pseudocode

Write pseudocode that will enable us to print the expected output above.

Step 2 – Implementation

Once you write pseudocode for this problem, implement the code using Java syntax.

Favorite Foods [Nested Collections Review]

Write a method called `favoriteFoods` that takes a `ratings` map indicating how each person rates various foods and a `minimum` rating and returns a map indicating all the foods each person has rated with at least the `minimum` rating.

The input map will have keys that are names (strings) and values which are maps with keys that are a food (strings) and values which are numbers in the range of 0.0 to 5.0 for the rating that person has given that food. An example would be if we had a variable called `ratings` that stored the following map in the format just described:

```
{"Matt"={"pie"=5.0, "ice cream"=5.0, "mushrooms"=0.0},
 "Kasey"={"chicken strips"=4.3, "cranberry sauce"=4.2},
 "Miya"={"lettuce"=2.4},
 "Nathan"={}}
```

In this example, we see that Matt has rated pie and ice cream as a 5.0 each and mushrooms as a 0.0, while Miya has only rated lettuce as a 2.4.

For example, suppose the following call is made:

```
favoriteFoods(ratings, 4.3);
```

Given this call, the following map would be returned:

```
{"Kasey"=["chicken strips"],
 "Matt"=["ice cream", "pie"],
 "Miya"=[],
 "Nathan"=[]}
```

Notice that the value for the key "Matt" is the set `["ice cream", "pie"]` because he rated only those foods with at least a rating of 4.3. The value for the key "Miya" is `[]` because Karen rated no foods with a rating of at least 4.3. Note that foods rated with a 4.3 should be included (see Kasey).

The map you return should have keys sorted alphabetically and the foods in the values should appear in alphabetical order as well. Your method should not modify the provided map. **You may assume that the map and none of its contents are null.**

Extra Java Review Resource

For folks who need an intro/refresher on Java syntax (e.g., those coming from a class that taught in a different language), click this link!

JUnit Practice - TFTPlayer

This example is a little bit more complex due to the rules of a game you might not be familiar with. You should read over the specification and try to write some tests as the task indicates, but we will pick up with this as an example in class to work on with your peers so it's okay if you don't feel super confident yet!

TFT is an eight-player competition where you are playing against the other players to be the last person standing. The game is pretty complicated with lots of different mechanics to add variance and skill expression to the game, but we will be focusing on a much smaller scope of the game to keep this problem as simple as possible.

Don't worry if you don't know this game or other games, we will provide a simplified set of background rules necessary to solve this problem and no outside experience is needed.

Rules of the Game

There are a lot of complicated rules and systems in TFT, but we will focus on the fundamental rules for attributes that players progress throughout the game: **gold, experience, and levels**. Every player has some amount of gold. They can spend gold to gain experience, and when they hit certain experience thresholds, they level up. Players passively gain gold based on an interest formula that rewards players for saving up gold before spending it.

Below are a simplified set of rules for this aspect of the game that we will use in our coding problem, **no outside rules or knowledge about the game should be needed figure out these mechanics**. If you do have experience with the game, note that the rules we are presenting are simplified to make this problem more manageable to code up.

- **Start of game**: Every player starts with 10 gold, 0 experience, and are level 1.
- **Buy Experience**: Players can spend 4 gold to gain 4 experience.
  - **Leveling Up** (Simplified): Every time a player reaches 20 experience, they level up (level increases by 1). The maximum level is 9 at which point the player can't purchase any more experience.
- **Gain Gold** (Simplified): Every turn the player receives some *interest* based on their current gold in addition to every player receiving 1 additional gold "for free". So in short, when gaining gold every player receives some amount of interest (described next) plus 1 additional gold.
  - **Interest:** The interest rule works as follows. Each player receives one extra gold in interest for every 10 gold they have. There is no additional interest for having more than 50 gold, so the maximum interest is 5 gold per turn. Here are some examples for gold being gained:
    - If a player has 24 gold currently, they would gain 3 gold (2 for their interest, and the 1 gold "for free"). The interest is 2 in this example because the player has 20 gold to earn 2 interest, but not 30 gold to earn 3.

        - If a player has 39 gold, they would gain 4 gold (3 for interest, 1 "for free").
        - If player has 55 gold, they would gain 6 gold (5 for interest, 1 "for free").
        - If a player has 72 gold, they would gain 6 gold (5 for interest, 1 "for free").

That's a lot of rules, but we wanted to spell it out clearly to make sure the scope of the problem was unambiguous.

Task

We have already written a `TFTPlayer` to represent each player in the game with the following methods to implement the rules of the game described above. Note that our implementation is potentially buggy. Your task is to write JUnit tests to test the behavior of this class and hopefully expose the presence of these bugs.

Task: Write at least one JUnit test for each of the methods. Ideally you'll have multiple tests for each method to test out different circumstances to ensure good test coverage. Because you aren't expected to fix the bugs for this problem, it is okay if some of your tests are failing due to our incorrect implementation.

- *Optional:* You can fix the bugs if you would like, but that is not the focus of our exercise. When we release the solutions, you can see an explanation of all the bugs present and how to fix them.

`TFTPlayer` class

- **`public TFTPlayer()`**

  Sets up a `TFTPlayer` with the start of game state (10 gold, 0 experience, level 1).
- **`public int getGold()`**

  Returns the current amount of gold
- **`public int getXP()`**

  Returns the current amount of experience
- **`public int getLevel()`**

  Returns the current level of the players
- **`public boolean buyXP()`**

  Spends 4 gold of the player's gold and increases their experience by 4 points. Returns `true` if the player was able to purchase experience and `false` otherwise. Some important cases to consider:
    - If the player has less than 4 gold, no gold should be spent and no experience should be

    gained.
  - The player can't purchase experience if they are already the maximum level (9).
  - If the player accumulates 20 experience points, those 20 experience points they should "level up" and gain a level in exchange for those 20 experience points.
- **`public int gainGold()`**

  Increases the amount of gold using the rule described above of adding `interest + 1` gold to the player's gold. Returns the new amount of gold the player has after this operation.

ClockTime [OOP Review]

Suppose that you are provided with a pre-written class `ClockTime` as shown below.

Write an instance method named `advance` that will be placed inside the `ClockTime` class to become a part of each `ClockTime` object's behavior. The `advance` method accepts a number of minutes as its parameter and moves the `ClockTime` object forward in time by that amount of minutes. The minutes passed could be any non-negative number, even a large number such as 500 or 1000000. If necessary, your object might wrap into the next hour or day, or it might wrap from the morning ("AM") to the evening ("PM") or vice versa. A `ClockTime` object doesn't care about what day it is; if you advance by 1 minute from 11:59 PM, it becomes 12:00 AM. For example, if the following object is declared in client code:

```
ClockTime time = new ClockTime(6, 27, "PM");
```

The following calls to your method would modify the object's state as indicated in the comments.

```
time.advance(1); // 6:28 PM
time.advance(30); // 6:58 PM
time.advance(5); // 7:03 PM
time.advance(60); // 8:03 PM
time.advance(128); // 10:11 PM
time.advance(180); // 1:11 AM
time.advance(1440); // 1:11 AM (1 day later)
time.advance(21075); // 4:26 PM (2 weeks later)
```

Assume that the state of the `ClockTime` object is valid at the start of the call and that the `amPm` field stores either "AM" or "PM".

This implementation of ClockTime with a hour, minute, and amPm field is only one of the many possible implementation! Try thinking about other possible selections of fields and how that would change the implementations of the other method!

USCurrency [OOP Review]

Define a class USCurrency that can be used to store a currency amount in dollars and cents (both integers) where one dollar equals 100 cents. Your class should include the following methods:

```
USCurrency(dollars, cents) -- constructs a currency object with given dollars and cents

dollars() -- returns the dollars

cents() -- returns the cents

toString() -- returns a String in standard $d.cc notation (-$d.cc for negative amounts)

add(other) -- returns the result of adding another currency amount to this one

subtract(other) -- returns the result of subtracting another currency amount from this one

compareCurrencies(other) -- returns an integer value that indicates if this currency is greater tha
```

A currency amount can be negative. The cents method should return values in the range of 0 to 99 for nonnegative currency amounts and should return values in the range of 0 to -99 for negative currency amounts.

The add and subtract methods should return new `USCurrency` objects that represent the result of adding or subtracting the other currency amount.

Note that the `toString` method should return the amount in a standard format ($d.cc) with two digits for cents and with negative values indicated with a single minus sign in front of the dollar sign (-$d.cc). For example, 4 dollars and 5 cents would be expressed as "$4.05" while -19 dollars and -43 cents would be expressed as "-$19.43".

The `compareCurrencies` method should return an integer indicating how *this* currency compares to the *other* currency. Smaller currency amounts should be considered "less" than larger currency amounts. (e.g., -$13.45 < -$2.03 < $5.13 < $98.06).

It should return:

- **1** if the amount of *this* currency is **greater** than the *other* currency
- **-1** if the amount of *this* currency is **less** than the *other* currency
- **0** if the amount of *this* currency is **equal** to the *other* currency

Have fun!

[SCAFFOLD] Big slide

This code slide does not have a description.

ADDITIONAL RESOURCES

The rest of the slides in this section are additional resources for you to look at if you need a refresher on OOP and Junit!

JUnit CheatSheet

Learning a new concept can be overwhelming so we've compiled a cheatsheet you can reference while you write your own JUnit tests!

Creating a JUnit Test Class and JUnit Test Case

To create a JUnit test class, make sure you import `org.junit.jupiter.api.*` and `static org.junit.jupiter.api.Assertions.*`. These will give you access to method annotations like `@Test` and `@BeforeEach` and assertion methods like `assertTrue()` and `assertFalse()`.

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class ExampleTestClass {
    @Test
    public void yourTestCase() {
        // Assertion methods called here
    }
}
```

JUnit Method Annotations

Method annotations are used by JUnit so that JUnit knows how to treat your methods. Before you write a method, you must attach a method annotation. This special syntax is then interpreted by JUnit to know how to execute your method.

`@Test`: Turns a public method into a JUnit test case.

```
@Test
public void test() {
    ...
}
```

`@Timeout(time)`: Times the test such that the test will fail after `time` milliseconds. Thus, the code must finish execution before `time`. Note that you still need the `@Test`.

```
// Test will fail after 1000 ms
@Test
@Timeout(1000)
public void test() {
    ...
}
```

`@BeforeEach`: The method will be executed before each `@Test`

```
private int num; // This is a field

// This method will execute before each @Test
@BeforeEach
public void setUp() {
    num = 0;
}

@Test
public void test() {
    assertSame(0, num);
    num++;
    assertSame(1, num);
}
```

JUnit Assertion Methods

Assertion methods are the building blocks of JUnit and how you will write testing code inside your test methods. When you use an assertion method, the result needs to match up with what the assertion method expects, otherwise, your test will fail. Below are the most common types of assertion methods that you will use:

`assertTrue(test)`: Fails if the `test` is `false`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertTrue(true);
    assertTrue(x == 2);
    assertTrue(s.equalsIgnoreCase("Hello World"));
}
```

`assertFalse(test)`: Fails if the `test` is `true`

```
@Test
public void test() {
    int x = 2;
    String s = "Hello World";
    assertFalse(false);
    assertFalse(x != 2);
    assertFalse(s.contains("a"));
}
```

`assertEquals(expected, actual)`: Fails if the `expected` and `actual` are not equal

```
@Test
public void test() {
    String s1 = "Hello World";
    String s2 = "Hello World";
    String s3 = "Hello World";
    assertEquals(s1, s2);
    assertEquals(s2, s3);
    assertEquals(s3, s1);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    for (int i = 1; i <= 5; i++) {
        list1.add(i);
        list2.add(i);
    }
    assertEquals(list1, list2);
}
```

`assertSame(expected, actual)`: Fails if the `expected` and `actual` are not equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertSame(2, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = list1;
    assertSame(list1, list2);
}
```

`assertNotSame(expected, actual)`: Fails if `expected` and `actual` are equal using reference semantics (==)

```
@Test
public void test() {
    int x = 2;
    assertNotSame(3, x);

    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();
    assertNotSame(list1, list2);
}
```

`assertNull(value)`: Fails if `value` is non-null

```
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNull(map.get("Hello World"));

    String s = null;
    assertNull(s);
}
```

`assertNotNull(value)`: Fails if `value` is null

```java
@Test
public void test() {
    Map<String, Integer> map = new HashMap<>();
    map.put("cse122", 1);
    assertNotNull(map.get("cse122"));

    String s = "Hello World";
    assertNotNull(s);
}
```

`assertArrayEquals(Any[] expectedValues, Any[] actualValues)`: Fails if `expectedValues` and `actualValues` do not have the same elements, in the same order.

```java
@Test
public void test() {
    int[] a = new int[] {1, 2, 3};
    int[] b = new int[] {1, 2, 3};
    assertArrayEquals(a, b);
}
```

`assertThrows(exception.class, () -> {code})`: Fails if `code` does not throw `exception`

```java
@Test
public void test() {
    List<Integer> list = new ArrayList<>();
    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.get(2); // List is currently: []
    });

    assertThrows(IndexOutOfBoundsException.class, () -> {
        list.add(1); // List is currently: [1]
        list.add(2); // List is currently: [1, 2]
        list.add(3); // List is currently: [1, 2, 3]
        list.remove(3); // Index 3
    });

}
```

Using JUnit to test Java's ArrayList Implementation:

Below is an example of a JUnit testing class that tests Java's ArrayList implementation:

```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import java.util.*;

public class ArrayListTest {
    private static final int TIMEOUT = 2000;
    private List<String> list;

    @BeforeEach
    public void setUp() {
        list = new ArrayList<>();
    }

    @Test
    @Timeout(TIMEOUT)
    public void testAddingElements() {
        assertTrue(list.isEmpty());
        list.add("Hunter Schafer");
        list.add("Miya Natsuhara");
        list.add("CSE 122");

        assertEquals("Hunter Schafer", list.get(0));
        assertEquals("Miya Natsuhara", list.get(1));
        assertEquals("CSE 122", list.get(2));

        assertTrue(list.size() == 3);
    }

    @Test
    public void testContains() {
        assertTrue(list.isEmpty());
        list.add("CSE 122");

        assertTrue(list.contains("CSE 122"));
        assertFalse(list.contains("Hello World"));
    }

    @Test
    public void testNegativeIndexGet() {
        assertTrue(list.isEmpty());
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(-1));
    }
}
```

⭑JUnit: Line

Suppose that you are provided with a pre-written class `Line` which implements the following behavior:

- `public void addPoint(Point other)`
  - Extends the current line to include the provided 'other' point. Each point is only counted once (duplicates are ignored).
- `public Point getStart()`
  - Returns the first Point in the line (the Point with the smallest x-value). Throws a `NoSuchElementException` if there are no points currently within the line.
- `public Point getEnd()`
  - Returns the last Point in the line (the Point with the largest x-value). Throws a `NoSuchElementException` if there are no points currently within the line.
- `public int getNumPoints()`
  - Returns the total number of points within this line segment.

Fill in the provided `Testing.java` following the instructions step-by-step to complete your testing suite.

NOTE: This is an example of "clear-box" testing, where you know and can view the current solution directly. This can be beneficial as after implementing a solution, programmers will likely be more knowledgeable about potential edge cases that would be useful to account for in their tests!