

WES-CS GROUP MEETING #3

Exercise 1: What's in a name? (Java Identifiers)

Being able to give things specific names makes it far easier to communicate instructions. In Java we can give things names so long as naming conventions are followed. Let's try to apply those naming conventions by playing a game. Divide into groups of two or three. Each group will get cards that say *valid*, *valid but poor*, or *invalid*. Your Team Leader will write a name that's supposed to be a Java identifier, and your group should decide which category it belongs in and why. Put the card that corresponds to your answer face down on the table in front of you. When all groups have placed their cards, reveal your answers and determine the correct answer (keep score if you wish).

Exercise 2: More Fun with Words (Tracing Algorithms)

Last week we did an exercise with an object called `word`. We'll use the `word` object again today, with a similar list of operations:

```
moveToFront( int pos )
```

Move the letter in position `pos` to the beginning of the word (i.e., to position zero).

```
moveToEnd( int pos )
```

Move the letter in position `pos` to the end of the word.

```
swap( int pos1, int pos2 )
```

Swap the letter in position `pos1` with the letter in position `pos2`.

```
reverse( int start, int finish )
```

Reverse the order of the letters in positions *start* to *finish*.

In Java, these operations are called *methods*, and code like `word.swap(0, 1)`, is called a *method call*: the `swap` method of `word` is being called to rearrange the letters in the sequence of letters that `word` represents.

Part (a).

First, make sure you remember what the `word` operations do; work with a partner to fill in the third column below. Remember that the first letter of the word is in position zero.

Original word	Java code	word after the code executes
HIPSH	<code>word.moveToFront(2)</code>	
ZOOLOGY	<code>word.moveToEnd(0)</code>	
PICKLES	<code>word.swap(0,6)</code>	
AVOCADO	<code>word.reverse(4,6)</code>	

Part (b).

The process of keeping track of what happens when a piece of code is executed is called *tracing*. Tracing can be tedious, but it is a valuable skill that helps programmers understand and fix code. We'll do some tracing of method call sequences for practice.

Work in groups of two, racing against the other groups. Your Team Leader will give each group some cards with letters on them. Start by arranging them to spell **debit-card**. Then you'll get a sequence of method calls. Your job is to rearrange your letters to show the effect of each call. When you have the final result, show it to your Team Leader. Don't say it out loud or the other teams might hear you!

Part (c).

Now your Team Leader will give each group a new starting value for `word` and a worksheet. The worksheet will have a new sequence of method calls, and space to keep track of the value of `word` after each call. Carefully record the execution trace by writing down the change that each method call makes to the word.

Part (d).

Find an anagram for your partner's name (you'll get a more interesting one if you use both the first and last name). You can invent the anagram yourself, or use one of the laptops to try out the anagram server at

www.wordsmith.org/anagram

Assume that `word` initially represents your partner's name (with no space between the first and last name). Write down a sequence of method calls that transforms it into your anagram. For a challenge, try to include at least one loop in your code. When you're done, give your sequence to your partner so that they can use it to discover your anagram.

Exercise 3: Baking Java Style (Methods that Return Values)

So far, the method calls we've used all performed actions. A method can also compute and return a value. For this exercise, we'll assume we have three objects named `brownies`, `cookies`, and `fudge`, each of which represents the recipe for a chocolate dessert. Each of these objects has the methods listed below. These methods compute and return integer values, so the descriptions all start with the special word `int`.

```
int numEggs()
```

Returns the number of eggs needed to make 1 batch of this dessert.

```
int numCupsFlour()
```

Returns the number of cups of flour needed to make 1 batch of this dessert.

```
int numOzChocolate()
```

Returns the number of ounces of chocolate needed to make 1 batch of this dessert.

If we want to know how many eggs are needed to make 1 batch each of brownies, cookies, and fudge, we could evaluate the following Java expression:

```
brownies.numEggs() + cookies.numEggs() + fudge.numEggs()
```

We could also store the result (an integer value) in a new Java variable by writing code like this:

```
int numEggsNeeded = brownies.numEggs() + cookies.numEggs() +  
                    fudge.numEggs();
```

Part (a).

What Java code could we use to find out how many eggs are needed to make *two* batches of fudge, and to store that number in a new variable?

How about the number of cups of flour needed to make two batches of fudge and three batches of brownies?

Part (b).

Now let's assume that we have an object called `onHand` that represents the ingredients we have in the house. The `onHand` object has the following method:

```
int numUnits( String ingredientName )
    Returns the number of units (cups, ounces, or whatever is appropriate) of the given
    ingredient that we have in the house.
```

For example, the expression `onHand.numUnits("eggs")` tells us how many eggs are in the house.

To make our desserts, we also need an object called `chef`, with a `bake` method. That method has two arguments: the recipe to bake and the `onHand` object to provide the ingredients. For example, to make a batch of brownies we'd use this code:

```
chef.bake( brownies, onHand );
```

Executing this code would cause the chef to bake a batch of brownies, and also to change the amount of ingredients in the house (by removing the ones that got used for the brownies).

Suppose we want to make one batch of fudge. We have plenty of flour and chocolate in the house, but we're not sure about eggs. How could we do the following:

1. Find out how many eggs are needed, and store that value in a variable.
2. Compare the number of eggs needed with the number of eggs in the house. If there are enough eggs in the house, bake one batch of fudge.

Hint: Use a Java *if-statement* or write an algorithm that uses *if* control flow.

Part (c).

Now suppose we want to make as many batches of cookies as possible. This time, chocolate is the ingredient in short supply. What Java expression would tell us how many batches of cookies we can make? (Hint: If we have N ounces of chocolate in the house, and each batch of cookies requires M ounces of chocolate, then what expression gives the number of batches of cookies we can make?)

Part (d).

Write a Java *while-loop* or write pseudo code that uses a loop control flow to cause the right number of batches of cookies to be baked. (Hint: You can either figure out how many batches are possible, like in Part (c) and use a loop to tell the chef to bake that many batches, or you can use a loop to bake one batch of cookies over and over as long as there is still enough chocolate in the house).

Exercise 5: Logical Thinking

Assume that you have 8 coins, and you know that 7 are OK but one is bad. You know that the bad coin has a different weight than the good coins, but you don't know whether it's heavier or lighter.

Figure out how, using only a balance scale, you can find out which is the bad coin using just 3 weighings. Hint: Find a way to determine that half of the coins are OK with just 1 weighing.

Now do the same thing assuming that you have 9 coins, one of which is bad. (Still use just 3 weighings to find the bad coin.)

And now for a real challenge, do the same thing assuming that you have 13 coins.

Identifier Exercise

1. Provide teams the answer cards and decide if they will keep score (see scoring below).
2. Have each team submit one identifier that they think is tricky.
3. Write the following identifiers, one at a time, on the board. Mix in the ones submitted by the teams.

xyz	MAX_VAL
This Is Fine	PB&J
WillThisWork?	abiggreenpickle
public	true
Public	three#s
letterCount	3people
big_bear	mendota22
fire-fly	everything's_\$1

Suggested scoring:

- 1 point** to each team for each correct answer (excluding their own submission)
- 1 point** bonus for each *valid but poor* identifier, to the first team that gives a correct reason why it is poor
- 1 point** bonus for each *invalid* identifier, to the first team that gives a correct reason why it is invalid
- 4 points** if a team gets its own submission wrong
- 2 points** to the team that submits an identifier, for every other team that gets it wrong (i.e., 2 points for every team that gets fooled)

Tracing Exercise Part (b)

d d d d

e e e e

b b b b

i i i i

t t t t

Tracing Exercise Part (b)

■

■

■

■

c

c

c

c

a

a

a

a

r

r

r

r

d

d

d

d

Tracing Exercise Part (b)

word.moveToFront(7)
word.moveToEnd(5)
word.swap(2,5)
word.reverse(3,6)
word.swap(7,4)
word.moveToEnd(5)
word.swap(8,9)
word.moveToFront(5)

word.moveToEnd(5)
word.swap(2,5)
word.reverse(3,6)
word.swap(7,4)
word.moveToEnd(5)
word.swap(8,9)
word.moveToFront(5)

word.moveToFront(7)
word.moveToEnd(5)
word.swap(2,5)
word.reverse(3,6)
word.swap(7,4)
word.moveToEnd(5)
word.swap(8,9)
word.moveToFront(5)

word.moveToFront(7)
word.moveToEnd(5)
word.swap(2,5)
word.reverse(3,6)
word.swap(7,4)
word.moveToEnd(5)
word.swap(8,9)
word.moveToFront(5)

word.moveToFront(7)

Tracing Exercise Part (c): 2 copies of this page per group

Starting value of word: dormitory_

Method Call	Value of word after the call
<code>word.moveToFront(4)</code>	
<code>word.moveToEnd(2)</code>	
<code>word.moveToFront(2)</code>	
<code>word.reverse(0, 2)</code>	
<code>word.moveToEnd(5)</code>	
<code>word.swap(5, 7)</code>	
<code>word.reverse(5, 6)</code>	
<code>word.moveToEnd(3)</code>	

Starting value of word: dormitory_

Method Call	Value of word after the call
<code>word.moveToFront(4)</code>	
<code>word.moveToEnd(2)</code>	
<code>word.moveToFront(2)</code>	
<code>word.reverse(0, 2)</code>	
<code>word.moveToEnd(5)</code>	
<code>word.swap(5, 7)</code>	
<code>word.reverse(5, 6)</code>	
<code>word.moveToEnd(3)</code>	

VALID

INVALID

VALID BUT POOR

VALID

INVALID

VALID BUT POOR

VALID

INVALID

VALID BUT POOR