# NEUTRINO

# Professional Java Developer Career Starter: Java Foundations

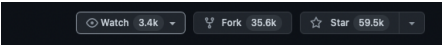Exercises & Supplements

# Table of Contents

# Preface

The exercises in this document range in difficulty, but some of you may find them to be quite difficult. One reason for this is that, I've tried to create exercises that may cause you to have to think more indirectly. For example, I could say, "Write a program that stores ten words in an array, generates an integer between zero and nine, and then uses that integer to select a random word out of the array." Or, I could say, "Write a program that randomly picks a word out of ten choices." The second sentence is better IMO because it's the way non-programmers would state something and forces you, the programmer, to translate that request into programmer mechanisms you know about. The first sentence basically tells you how to solve the problem already.

The part of programming that I think will be hardest for me to teach you is how to translate real-world requests into code that is comprised of fundamental building blocks of programming. However, I find that part to be the most fun because that's where you can use your creativity. Some of you will think every problem you may be asked to solve, already has a known solution and your job as a programmer is to simply memorize all the solutions and use one when needed. Of course, to a certain degree, this is partially true because there are very few new requests anyone will ask of a programmer these days. However, the likelihood of you knowing all those solutions is low. And, the time it may take you to find them with Google may take longer than using your creativity to create your own approach. You will have to gauge when it's appropriate to do one or the other. I can tell you that all the challenges (exercises) in this document, are meant to be within your own abilities and you should not need to Google for how to do them (well, you may need/want to Google for syntax or specific methods whose names you don't recall, but not for general approach/strategy/algorithm) - but many will be hard to do for some of you. I've tried to include hints or notes (which you find in the left margin of this document - so don't look if you don't want to cheat ;-) ) for those problems when you need more help, but try to do them on your own, without the notes first. If, after reading the notes, you still can't figure out a way, I've also provided fully-worked solutions for nearly every problem (only left out a few that I felt were obvious). Those solutions can be found in here in GitHub (it's clickable). By the way, I may frequently update the solutions project in GitHub. So, if you'd like to receive automatic email notifications whenever I do update it, you can click the "Watch" button in the upper right top of the project's repository page in GitHub.



You may notice from section to section, I may give exercises that call on you to use skills you learned in previous sections. This is intentional so that you can get more opportunities to practice putting all these techniques

**NEUTRINO**
© 2021 by Neutrino Systems Inc.

together to solve bigger problems. I may also repeat exercises in later sections but require you to solve them again using newer techniques that you've now learned.

Finally, let me say that these exercises are here for you to do as you will. Don't think of them as homework. I don't care if you do them or not. Some of you won't do any, that's perfectly fine with me. But, you will become much better if you practice, and that's what these are for. So, for those of you who choose to try them out… good luck and HAVE FUN!

# Classes, Object & Methods

## Supplemental on Arrays

Arrays can also be multi-dimensional. In the video lecture, I only show how to create arrays of one dimension, which can be thought of as one column of values in a spreadsheet, or one row. But arrays can model a table of two or more columns and/or rows, or have even more dimensions than that.

| One Dimension | | |
|---|---|---|
| one | two | three |

| Two Dimension | | |
|---|---|---|
| one | two | three |
| a | b | c |

To create a two-dimensional array, you can do:

```java
String[][] my2d = new String[2][3];
my2d[0][0] = "one";
my2d[0][1] = "two";
my2d[0][2] = "three";
my2d[1][0] = "a";
my2d[1][1] = "b";
my2d[1][2] = "c";
System.out.println(my2d[0][2]);
```

Or, alternatively, you can declare and initialize the array at the same time like so:

```java
String[][] my2d = {{"one", "two", "three"}, {"a", "b", "c"}};
System.out.println(my2d[1][0]);
```

## Exercises

✓ 1. Create a new Java project and try modeling a domain you're comfortable with. It could be something like: Sports, Business, Mathematics, Online Shopping, etc. For example, if you modeled online shopping, you might create classes to represent: Customer, Product, Order, etc. These classes would likely have properties and you may think of a few methods that might make sense for them too. There is no example code for this.

✓ 2. Create an array and initialize it with the days of the week.
   2.1. Use System.out.println() to print:
      2.1.1. The number of items in the array

    2.1.2. The 4th day in the array

3. Create an array and initialize it with numbers, 1-10.
4. Try creating an array to represent the tic-tac-toe board to the right.
    4.1. How would you access the value in the bottom right square with Java code?
5. Write a method that allows you to pass any number of Strings as parameter inputs without using an array.
6. How can you create a method that can be called without creating an instance of its class?
7. Model a Car, create an instance of it, pass it to System.out.println() so that its properties will be printed.
8. Model a Bank with the ability to access the vault. Model a BankManager and a Customer too. Ensure the Customer can not access the Bank's vault directly but the BankManager can.
9. How can you ensure that a Customer instance can not be created without a name and an initial deposit?
10. How might you model a Math class to know about Euler's number?
11. Model a class with data that enumerates (associates with a number) the first names of three friends, classmates or coworkers. Make it so that this name data will be shared among all instances of the class.

## Working with Text

### Exercises

1. Model a Person with a first name and last name and ensure that even if the first name is entered all lowercase, it will be stored all uppercase.
2. Write code that replaces the word "cat" with the word "dog" wherever it shows up in a sentence.
3. How can you make sure when people enter text into a program, there are no unintended spaces at the beginning or end of the text?
4. If someone enters "   alphabet " (notice the spaces) as input into a method, make that method output "alphabeT" instead.
5. Given the String "12345 Big St., Alphabet City, CA 90210" or any other address with the same format, can you write code that can parse and print out:
   1. The building number: 12345
   2. The Street: "Big St."
   3. City: "Alphabet City"
   4. State: "CA"
   5. Postal Code: 90210

# Regular Expressions

Supplemental on Regex

**Backreferences**

Java Regular Expressions also support *backreferences*. This is a capability of a regex to refer back to a capture group from earlier in the expression. For example - if given a string:

"aaabbb12345cccdddzzz12345"

We could write a regex that matches the same sequence of numbers like so:

"[a-z]+(\\d{5})[a-z]+**\\1**"

Where:

"[a-z]+" matches "aaabbb"

"(\\d{5})" matches "12345" AND is stored in reference #1

"[a-z]+" also matches "cccdddzzz"

"\\1" refers back to capture group #1 which is "12345"

We can also use named capture groups with backreferences like so:

"[a-z]+(?<nums>\\d{5})[a-z]+**\\k<nums>**"

Note the use of "\\k<nums>" to refer back to the named capture group of "nums". Be sure to include the "k" in "\\k" to indicate you're backreferencing a named group.

## Exercises

1. Write a regex that would match the following words: Dark, bark, Lark
    1. For extra challenge, could you additionally make it match: stark
2. Use capture groups to write a regex that could match: **A**bracada**bra** or A**gra**cada**gra**
3. How can you use parentheses in a regex for grouping but without capturing?
4. Write a regex that tests whether a String is an address and allows you to extract the parts (your choice for address format).
5. Write a regex that tests whether a String is an email address.
    1. Note: Doing this in regex is actually notoriously difficult to comply with **ALL** the ways an email address can be written. However, you can just do the simplest form of email address you can think of, such as: first.last@domain.com

## Comments (left margin)

**1. Terry Martin**
January 7, 2022 at 5:20:05 PM
You could assign each type of signal to a bit of one byte. For each type of signal received, turn on its corresponding bit. Then, convert the binary bit pattern to a number.

**2. Terry Martin**
January 7, 2022 at 5:17:36 PM
Take the binary bit pattern representing the signal types received, and AND it with the binary bit pattern of the signal type you're testing for.

If you received 1 0 1, and you want to know if the leftmost bit is a 1, you'd AND
1 0 1 AND
1 0 0
The result would be:
1 0 0
And you'd know the leftmost bit was "on".

**3. Terry Martin**
January 7, 2022 at 5:25:03 PM
You can use a logic OR operation. In Java, this is the pipe symbol, " | ".
2 + 4 is equivalent to:
2 | 4

Note: this only works for numbers whose binary bit patterns don't overlap.
2 | 2 does not equal 2 + 2
0 0 0 0 0 0 1 0 = 2
       OR
0 0 0 0 0 1 0 0 = 4
— — — — — — — —
0 0 0 0 0 1 1 0 = 6

BUT
0 0 0 0 0 0 1 0 = 2
       OR
0 0 0 0 0 0 1 0 = 2
— — — — — — — —
0 0 0 0 0 0 1 0 = 2

**4. Terry Martin**
January 8, 2022 at 3:14:11 PM
You'll need a way to store the 10 names in a class.

You'll need a way to access each name each time the next() method is called.

Consider using an array, a number and incrementing the number.

**5. Terry Martin**
January 8, 2022 at 3:46:35 PM
This problem is mostly about simple math and your ability to recognize a simple pattern. When you have a range of data in a set that needs to be classified or grouped at regular intervals, you're often looking at dividing by that regular interval. In this case, the

---

# Numbers

## Exercises

1. [1] If you could receive up to eight different types of radio signals simultaneously (into your computer & a program you wrote) and you needed to be able to record which of those eight you received at any given time, what would be a highly compact (or compressed - using the least amount of memory or storage) way to record them (using what you learned in this module)? For example, you could receive signals A + B + C simultaneously, or A + D, or just G, etc.

   1. [2] If you received signals A + D + C simultaneously, how could you efficiently determine that D was one of the signals you received - based on your solution above?

2. [3] What's an alternative way you could add 2 + 4 without using "+" symbol?

3. [4] Using only what you've learned so far in this course, create a class that contains 10 lowercase names and has a method that can be called 10 separate times, each time, returning the next name. If the method is called "next()", the first time it's called, it returns "name1", the second time "next()" is called, it returns "name2", etc.

   1. Make the next() method capitalize the first letter of each name as it outputs them.

4. [5] Imagine you've been given data representing how long items have sat in a warehouse. Your job is to categorize them by time in the warehouse. Items can be classified as 0, 1, 2 or 3. Items in class 0 are less than 89 days old. Class 1 = 90-179 days, Class 2 = 180-269 days, Class 3 = 270-364 days. For the given warehouse wait times: 13, 49, 90, 130, 175, 181, 255, 310, 330, 359, write a class similar to Exercise 3 above, that has a next() method that can be called 10 times and outputs a number representing the classification of each number in the set of ages above. Example: an item waiting for 5 days would return 0 and an item waiting 92 days would return 1.

5. Make a method, next(), that can be called 10 times and generate a random integer between 0 & 10 (non-inclusive). This method keep a running sum of all random numbers it generates and return that sum each time. If first time next() is called, it generates 5, and second time it's called it generates 3, it should return 8 from the second call - for example.

6. Write a function that takes a String like "149.32" and formats it as money for wherever you live. So, if you live in the United States, it would return "$149.32", Korea = ₩149, France/EU = 149,32 €, etc.

7. Write a function that takes a String input of "$12,345.83" and prints out that value divided by 32.19. It should return $383.53.

8. Use printf() to format the following inputs to print out the following outputs

interval is 90. By dividing each value by 90, you'll translate the values directly into their numerical classification of 0, 1, 2 or 3.

| Input | Output |
|---|---|
| 123456.783 | $123,456.78 |
| -9876.32532 | (9,876.325) |
| 23.19283928394829182 | 2.319284e+01f |
| 123456 | 0000123456 |
| -9876.35532 | -9,876.4 |

9. How could you format each of the inputs in the table above and store them in String variables instead of just printing them out directly?

10. Use instances of DecimalFormat to perform the same conversions in the table above.

11. Write a method that takes the String inputs, "37" & "13", and returns an integer of their sum, 50.

# Control Flow

## Exercises

1. Write a method in 4 lines of code (not including method name/signature & curly braces) that outputs the days of the week, one per line, using a regular 'for-loop' and an array.
   1. Do it again using the enhanced 'for-loop'.
   2. Do it again but make every other output line fully capitalized.
   Sunday
   MONDAY
   Tuesday
   WEDNESDAY
   etc…
   3. Do it again but use only 4 lines of code (not including method and curly braces of method.
2. Repeat all parts of exercise #1, but use 'while-loop' instead of 'for-loop' (where applicable).
3. Do #2 again but use a 'do-while-loop'. (No solutions provided for this one)
4. Use an array of days of the week, enhanced 'for-loop' and if/else to create the following output.

   *We eat pot roast on Sunday*
   *We eat spaghetti on Monday*
   *We eat tacos on Tuesday*
   *We eat chicken on Wednesday*
   *We eat meatloaf on Thursday*
   *We eat hamburgers on Friday*
   *We eat pizza on Saturday*

5. Repeat exercise #4 but instead of if/else, use traditional switch/case.
6. Repeat exercise #5 but use newer switch expression with as little repetition as possible.
   1. Add a private method that can capitalize the first letter of each word of the meal. So, instead of "We eat spaghetti on Sunday", it would print, "We eat Spaghetti on Sunday". Do not just capitalize the meal names yourself. Let your new method do it for you.
   2. Improve on 6.1 to make it smart enough to output: "We eat Pot Roast on Sunday", i.e., it should capitalize each word of a multi-word meal name.
7. Iterate over all the days of the week in an array and add up the total number of characters in all the days of the week. E.g., "Sunday"=6, "Monday"=6, etc. therefore, answer should be 50.

8.  8.  Use Regex with named parameters and a loop to parse the addresses below and print out the street address, city, state & postal code. Transfer this list of addresses into your program using a Java text block """" String.

*12345 First Street, First City, AA 90210*

*22222 Second Street, Second City, BB 22222*

*33333 Third Street, Third City, CC 33333*

*44444 Fourth Street, Fourth City, DD 44444*

*55555 Fifth Street, Fifth City, EE 55555*

*66666 Sixth Street, Sixth City, FF 66666*

*77777 Seventh Street, Seventh City, GG 77777*

*88888 Eighth Street, Eighth City, HH 88888*

*99999 Ninth Street, Ninth City, II 99999*

*00000 Tenth Street, Tenth City, JJ 00000*

# Testing

## Exercises

1. Use TDD to write tests and a SUT (system under test - the actual implementing class) for a method that takes a String input and outputs that same String with every other letter upper-case. For example, if you enter "cat", it returns "cAt". If you enter "apple", it returns "aPpLe". You should end up with two classes - a class with the new method in it, and a class for the unit tests that test your new method. Your test class should contain as many test methods as necessary to test the proper implementation.

2. Use TDD to write tests for a method that can take a String input like:

   ***"Billy, Bob, 1234 Big St., Big City, California, 90210"***

   And return an Object representing that person. You can name your class Exercise2 or Person or whatever you like. To test this easily, you'll need to have the IDE generate an equals() & hashcode() method in your class that models a Person (you'll learn more about that in a later section). To do so, you can right click on an empty line within your class and select, "Generate…" and click "Next" every time (probably will be three times) followed by "Finish". You'll now be able to continue with your test(s).

   1. Modify your method to accept a String like:

   ***"Billy, Bob, 1234 Big St., Big City, California, 90210 | Joe,Smith,5678 Little St., Little City, New York, 20109"***

   Your new method should return an array of Objects that model a Person. You should reuse the same Person class you used before (whatever you named it). The text input should separate the details of people by the *pipe* symbol ' | '. There should be no limit on the number of people details you can pass in on one String. In the example String above, I present only two people, but you could keep adding a | symbol and more people details and you should get back an array with that many people objects in it. Try to reuse what you already did for the first part of this exercise and just build on top of it.

3. Use TDD to write a program that takes a String of two words and creates a "spoonerism" of them. A spoonerism is when the first letters or phonemes of two words are transposed. An example would be "**cr**ushing **bl**ow" -> "**bl**ushing **cr**ow" or "my bad" -> "by mad". Use these examples for test use-cases:

| BEFORE | SPOONERIZED |
|---|---|
| Crushing blow | Blushing Crow |
| Sound bite | Bound site |

| BEFORE | SPOONERIZED |
|--------|-------------|
| **Flat cap** | Cat flap |
| **Sad ballad** | Bad sallad |
| **Master plan** | Plaster man |
| **Trail snacks** | Snail tracks |

1. Can you make it work regardless of capitalization? (No example solution done for this one - you're on your own kid :-) )

# More OOP

## Exercises

1. Use an Enum to model the days of the week and print them using a loop.
    1. Print them with the first letter capitalized (without changing the Enum constants).
    2. Alternate between printing the first letter capitalized or whatever letter is approximately in the middle of the word. For example: **S**unday, mon**D**ay, **T**uesday, wedn**E**sday, etc.
    3. Print 10 randomly-chosen days of the week

2. Write code to print the following using only an Enum & loop without using any conditionals (if/else/switch/etc.)

    *We eat pot roast on Sunday*

    *We eat spaghetti on Monday*

    *We eat tacos on Tuesday*

    *We eat chicken on Wednesday*

    *We eat meatloaf on Thursday*

    *We eat hamburgers on Friday*

    *We eat pizza on Saturday*

    1. Do it again but capitalize the meal names too. "pot roast" should become "Pot Roast". (No solution provided because you've done this before in earlier exercise - but try to do it without looking at previous work - for practice)

3. Write a method that takes the following input String (in bold) and returns the following output String using an Enum like (print the output):

    getMealsForDays(**"friday, thursday, monday, saturday, tuesday"**)

    Output: hamburgers, meatloaf, spaghetti, etc…

4. Write a method that takes an input of 1 lowercase letter (and only 1 letter), like a - b - c - z, etc., and returns that letter's ordinal position in the alphabet, i.e. a = 1, b = 2, c = 3… z = 26. Do NOT use any conditionals (if/then/else/switch/etc.). Do not use an array. Write unit tests to test out the following inputs: a, z, w, f, c, h - which should return: 1, 26, 23, 6, 3, 8 respectively

5. Write a method that does the opposite of #4 - taking a number, 1-26, and returns a lowercase letter. 1 would return a, 2 = b, 26 = z, etc.

6. Write a method that takes the following inputs and returns the corresponding outputs:

    8->0, 7->1, 6->2, 5->3, 4->4, 3->5, 2->6, 1->7

7. If you modeled a chess board in Java using a two-dimensional array, how could you convert chess notation, or coordinate notation of chess, to an element in that two-dimensional array? For example, in the chess board image below, square a8 might map to

array[0][0], and h1 might map to array[7][7]. Make a method that takes an input String of a chess coordinate and returns array coordinates. This exercise simply builds on the previous two. Use the following test data:

| Input | Output |
|-------|--------|
| a8 | 0,0 |
| h1 | 7,7 |
| g5 | 6,3 |
| d4 | 3,4 |
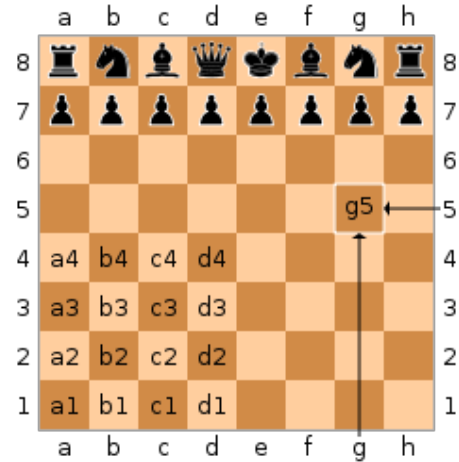


8. Now, let's do something **BIG**. Let's model a game of chess partially. Don't worry, we won't be modeling a full game and calculating the most efficient move. We will model the board and at least a couple of chess pieces, such as the pawn and knight. If you're not familiar with the basics of chess, please read this https://en.wikipedia.org/wiki/Chess#Movement. One of the places where Object-Oriented Programming really shines best, is in modeling domains with complex rules or logic. Few games can get as complex as chess. You'll be attempting to model how pawns and knights are permitted to move and interact with the board. I will provide JUnit tests that you can copy into your own project. You will then need to use TDD to implement the classes referenced in the tests and get the tests passing. For those of you who want to try modeling these exercises initially without any help at all, you may want to try not looking at the provided JUnit tests and make up your own based purely on the descriptions below. The provided tests will show you how I decided to model the concepts in some ways, though you will not see how their methods were implemented - that's left to you.

1. You'll need to create classes for: ChessBoard & Pawn. Per the JUnit test, ***canAddPawn()***, write code to allow an instance of the Pawn class to be added to an instance of the ChessBoard at a square on the board using chess notation (a1, c3, etc.). Then, assert that the pawn is actually at that location on the board by calling a method on the board to retrieve a piece by location.

2. Do the same as 8.1 but for a Knight, to get **canAddKnight()** working. The method you wrote in 8.1 to add a pawn should also be able to accept a knight. Please be mindful that you'll need to utilize Object-Oriented techniques you learned in this section to get this all working properly.

3. Switching to the JUnit tests in the PawnTest.java file, enable the **pawnCanMoveOneForward()** test to pass by implementing the necessary code. This is where you'll really begin to try implementing some business (or game) logic. The point of this test is to simulate creating a pawn and placing it on a square and "asking" it what are all the valid squares it could theoretically move to. The test will assert that one of its valid moves is to move forward by one square. These moves do not consider whether the square may already be occupied. Please be aware that the valid movements of a piece should always be relative to wherever that piece happens to start. So, although I provide examples of starting/ending positions for many of the following exercises, the movements should work consistently no matter where the pieces start. Moving forward from a2 to a3 is the same as moving from b2 to b3 or e5 to e6, etc. Do not hard-code the movements into your implementing code, in other words.

4. **pawnCanMoveTwoForwardOnFirstMove()** - hopefully, this test is obvious from its name. It asserts that of all the possible moves a pawn could make, if the pawn has never moved before (this assumes it begins on its standard starting position of a game), one of its allowed moves is to move two squares forward (again, ignoring possibility of other pieces in its way).

5. Get test **canNotMoveTwoAfterFirstMove()** passing. This test asserts that after a pawn has made its first move, if you "ask" it what its next valid moves are, moving two squares forward will not be one of its options.

6. Get test **canMoveOneDiagonallyRight()** passing. This asserts that one of a pawn's valid moves is to be able to move diagonally forward and to the right. Though this is only permitted when the pawn can capture a piece, we won't factor that in for now.

7. Get test **canMoveOneDiagonallyLeft()** passing. Same as 8.6 but to the left.

8. Get **blackPawnCanMoveForward()** passing. Assuming all previous pawn tests were for a white pawn moving "up" the board, that would mean that a black pawn would move in the opposite direction on the board. If we tie the direction of movement to the color of the piece, then this test will assert that the permitted movements of a black pawn will move "down" the board, towards the white pieces in general. If a black pawn is on square a7 and move forward (relative to its side of the board) by one square, it should then be on square a6.

9.  Get **blackPawnCanMoveTwoForwardOnFirstMove()** passing. Just the black pawn version of 8.4. If pawn was on b7, then it would end up on b5 in this case.

10. Switching to the KnightTest.java file, the first test to get passing is **knightCanMoveNorthEast()**. The idea here is to assert that one of the knight's valid moves is to move two steps forward (North) and one step right (East) from wherever it's beginning. So, if it starts on square c1, it would end on d3. I chose to include knights to be modeled because their movement is rather odd and considerably different from all other chess pieces. This may force you to really think about those Object-Oriented ideas ;-) .

11. **knightCanMoveNorthWest()** - from c1, knight would end up on b3.

12. **knightCanMoveEastNorth()** - from c1, knight would end up on e2.

13. **knightCanMoveEastSouth()** - from c3, knight would end up on e2.

14. **knightCanMoveWestNorth()** - from c3, knight would end up on a4.

15. **knightCanMoveWestSouth()** - from c3, knight would end up on a2.

16. **knightCanMoveSouthEast()** - from c3, knight would end up on d1.

17. **knightCanMoveSouthWest()** - from c3, knight would end up on b1.

18. **blackKnightCanMoveSouthEast()** - from d5, knight would end up on c7.

19. Now that you've taught your Pawn & Knight classes how they're permitted to move, let's enable them to be moved by the ChessBoard class. Going back to the tests for the ChessBoard, ChessBoardTest.java file (if you're going by the provided files), please implement the ability **canMoveC1KnightToD3()**. The test will create a ChessBoard and add a knight at c1. It will assert that the knight is on square c1. Then, it will then "request" the ChessBoard to move the knight to d3. It will then assert that the knight is no longer at c1 **and** that it is now at d3.

20. Per the **canNotMoveC1KnightToInvalidSquare()**, implement the ability to prevent the knight from being moved to invalid squares, such as from c1 to d4. In this event, the knight should simply remain at c1 (for now). Assert that the knight started at c1, then the attempt to move occurred. Then assert that the knight is **NOT** at d4 (as requested) but **IS** still at c1 in this case.

21. Per the **canNotMoveC1KnightToFriendlyOccupiedSquare()**, *implement the ability to prevent the knight from being moved to a square that is already occupied by another chess piece of the same color (a friendly). So, if a knight starts on c1, and there's a pawn starting at d3, the knight can not move() to d3 if the knight and pawn are both the same color.*

22. *Per the **canMoveC1KnightToEnemyOccupiedSquare(),** implement the ability to allow the knight to move to a valid square that is already occupied by an enemy piece (a piece of the opposite color). Assert that the piece you're moving is no longer at its starting position. Assert that it ends up at the destination square. Assert that the enemy piece has been "captured" by having been moved into an array of white's captured pieces.*

# Collections

## Exercises

1. Write code that allows you to model and store a collection of at least 5 cars and keeps them in the order in which they were entered. Print them out to the screen also.

2. Same as exercise 1 except we don't care about retaining the order **and** we want to ensure that duplicates will not exist.

3. Same as exercise 1 **but** associate an owner's first name with each car. Do not add owner's name to your car model class. Print out each owner with their car. Example:

| | |
|---|---|
| **Bob** | Car[make=Tesla, model=X, year=2015] |
| **Jenny** | Car[make=Tesla, model=Y, year=2016] |
| **Sarah** | Car[make=Tesla, model=3, year=2019] |

4. If you implemented exercise 2 with a record (instead of a class), do it again using a class or vice-versa.

5. Allow the cars from exercise 2 to be printed in "natural" order by model.
   1. [12] Store the same cars in a List and sort them. (No code solution provided)
   2. [13] Store the same cars in a List and sort them without implementing any interfaces on the Car class.
   3. How could you reverse the sort order?

6. Same as exercise 5 but allow program to remove a model by passing the model name as an argument to the main() method. For example, if you had been getting the following output initially:

Car[make=Tesla, model=3, year=2016]
Car[make=Tesla, model=Roadster, year=2009]
Car[make=Tesla, model=S, year=2014]
Car[make=Tesla, model=X, year=2015]
Car[make=Tesla, model=Y, year=2017]

And then you modify the program as instructed and pass in "Roadster" as an argument and re-run, you'd get the following output:
Car[make=Tesla, model=3, year=2016]
Car[make=Tesla, model=S, year=2014]
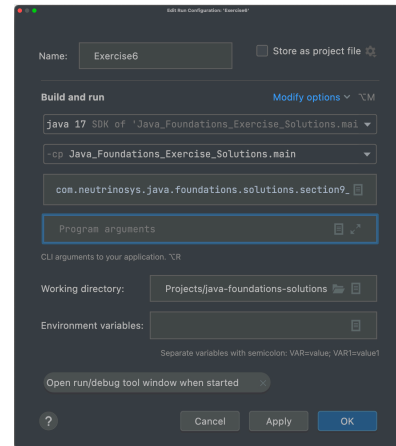Car[make=Tesla, model=X, year=2015]
Car[make=Tesla, model=Y, year=2017]

Instructions on how to pass arguments are below.
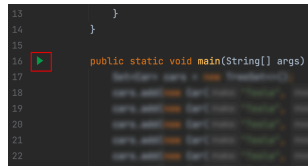


1. Right click the green triangle
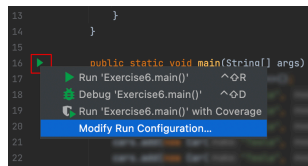


2. Select "Modify Run…"



3. Enter name of model to remove in "Program arguments" field & click "OK"

**14. Terry Martin**

February 7, 2022 at 10:59:37 AM

I'd strongly consider using a LinkedList. First, you need to retain the order in which they come (to you), so you'll need some type of List. Secondly, you're dealing with millions or billions. ArrayList would work BUT it pre-allocates large chunks of memory whenever it needs to make more space. This can be inefficient and wasteful of memory temporarily. A linked list can grow dynamically very efficiently.

**15. Terry Martin**

February 7, 2022 at 1:16:02 PM

The internal chess board two-dimensional array is the only array you should consider keeping. All others should be replaceable by a Collection. Additionally, all methods we explicitly wrote for adding to, removing from and searching an array, should be greatly simplified or removed because Collections classes have these capabilities built-in.

`14` 7. What might be a more memory-efficient collection to use to store millions or billions of objects in the order in which they come, if you don't know exactly how many objects you'll need to store before-hand?

8. How could you take your collection of cars from exercise 6 and store them in an array efficiently? How could you convert that array of cars back into a list again? (Solution is included in solution to exercise 6)

`15` 9. Now that you know about Java Collections, try revising the chess code from the exercises of section 8 to use collections instead of arrays (except for the internal chess board representation). *No code solution available at this time*

# Streams & Lambdas

## Exercises

1. Using the same car collection approach as in the last section's exercises, create a collection of cars but use the Streams API to print out only the cars' model names.

2. How could you use the Streams API to filter out (not show) all cars made before a certain year? (If you modeled Car year using the Year date/time class, it has a method *isAfter()*)

3. How could you create a stream of cars without first explicitly creating a collection of cars?

4. Add an additional integer price field to your Car class/record. Use Streams to add total cost of all cars in your collection/stream. (Two ways to do this)

   1. Use Streams to find average price of all cars (two ways to do this too) - *no code provided*

   2. How could you do exercise 4 with Streams & BigDecimal (for decimal accuracy)?

   3. How could you do exercise 4.2 but output a formatted money String still using only the Streams API? *Solution shown in 4.2 solution*

5. *Use the Streams API to sort a stream of cars in reverse order by model*

   1. *Do so by sorting by make, then model, then year - all in one*

6. *Use only the Streams API and a collection or Stream of Car objects to produce the following output: "S, X, 3, Y, Roadster"*

7. *Add more different makes of cars to your collection of cars and then*

| Make | Model | Year | Price |
|---|---|---|---|
| Tesla | S | 2014 | $90,000.99 |
| Tesla | X | 2015 | $110,000.99 |
| Tesla | 3 | 2016 | $55,000.99 |
| Tesla | Y | 2017 | $60,000.99 |
| Tesla | Roadster | 2009 | $135,000.99 |
| Lucid | Air | 2021 | $77,399.99 |
| Hyundai | Ioniq | 2021 | $34,250.00 |
| Hyundai | Kona | 2021 | $38,575.00 |
| Porsche | Taycan | 2021 | $81,250.00 |
| Audi | e-tron | 2021 | $66,995.00 |

**∧•NEUTRINO**

| Make | Model | Year | Price |
|---|---|---|---|
| **Volkswagen** | ID.4 | 2021 | $41,190.00 |

1. *Determine total price by make*
2. *Determine average car price by make*
    1. *This one will be fun/challenging. You'll want to take a look at the Collectors.teeing() function, which I did not explicitly teach but have a look at its javadoc. If this proves to be too hard but you want to still try it without looking @ my solution, it will be **MUCH** easier if you convert the prices to any other numeric type before using Streams API (but not as much fun…)*
    2. *Did you get 7.2.1? Let's make it even more **fun**. Format the average prices for currency, within the Streams API.*
3. *Determine number of cars by year and then by make*
4. *Determine number of cars by make by first creating a new, empty Map, then iterating over the collection of cars (you've been using for previous exercises) and using a functional method of the Map interface to populate your empty Map.*

# Loose Ends

## Exercises

17    1.  Revise the chess modeling from Section 8, exercise 8 to throw an exception when a chess piece tries to move to a square occupied by a friendly piece. Consider what is an appropriate type of exception for this. *No code solution provided*

2.  Try using Optionals in the chess modeling exercise instead of null values throughout the code.

3.  Create a collection of at least five Person objects with first name & last name fields. Add a few null objects to the collection in random places. Use an enhanced for-loop to iterate over the collection and print out the first names of each person. Use Optional to prevent NullPointerException and to print "Unknown" for the first names of null items.

    1.  Use the Optional API to display "Unknown" for null entries **and** entries whose first name is shorter than 3 characters.

4.  Assuming you captured user input of a date/time into a String like "July 12, 1984 13:47:00" and assuming that date/time occurred at GMT-8, write code to parse that String and convert it to GMT-0.

    1.  What would that same date/time be after 179 days, 7 hours and 27 minutes in Japan?

5.  Write code to determine the current age of your own country.

    1.  Write code to determine your own age exactly. *No code provided*

    2.  *Write code to determine how many days until the next New Year (or any holiday you choose). Note/hint: ChronoUnit has enums for various time intervals which also have useful methods.*

    3.  If you began an activity at 9:37:20 and stopped at 19:13:41, how best can you write code to determine how much time elapsed?

        1.  What if you want to know the elapsed time in minutes?

6.  How much will have elapsed if you depart from Seoul, South Korea at 13:15:00 February 3, 2022 and arrive in London, UK at 20:02:13, February 3, 2022?

    1.  When you arrive in London, what time will it be in Los Angeles, California?

7. Write code to parse each of these dates and print their corresponding date/time object. I provide code solutions for parsing 1-5 & 13 in the table here. The rest should be similar enough to those.

| |
|---|
| 2022-02-09T05:02:01Z |
| 2022-02-09T05:02:01+0000 |
| Wed, 09 Feb 2022 05:02:01 +0000 |
| Wednesday, 09-Feb-22 05:02:01 UTC |
| Wed, 09 Feb 22 05:02:01 +0000 |
| Wed, 09 Feb 2022 05:02:01 +0000 |
| Wed, 09 Feb 22 05:02:01 +0000 |
| 2022-02-09T05:02:01+00:00 |
| 2022-02-09T05:02:01+00:00 |
| Wednesday, 09-Feb-2022 05:02:01 UTC |
| Wed, 09 Feb 2022 05:02:01 +0000 |
| 2022-02-09T05:02:01+00:00 |
| 1644382921 |
| 2022-09-02 05:02:01 |
| 2022-09-02 05:02:01 AM |
| 09-02-2022 05:02:01 |
| 02-09-2022 05:02:01 |