Refactoring Activity 3

In this activity we will work with the PrimeGenerator class. Currently this class consists of one big method, and obviously we will want to break it down into smaller pieces. In order to do that, we'll need to understand the algorithm a bit more.

Step 9: Initial steps

For this file we will take a slightly different approach to what we did in the previous file. We will start by turning all the local variables into fields. This will allow us to freely break the algorithm up into smaller pieces. Then we'll see where that takes us, and hopefully eliminate some fields or turn them back into local variables.

- Go through each local variable and perform "Extract Field" on it. This includes primes, ord, square, candidate, primeIndex, ordmax, multiple, possiblyPrime.
- Now we break our code up into smaller chunks. Let's start with the field initializations near the top of the generate method. Move that section into the constructor, and run your tests to make sure we did not break anything.
- The comparison in the big while loop test looks to see if we need to compute more primes. Let's extract that into a method needMorePrimes. Don't accept the signature change.
- Now the whole body of that while loop can be extracted to a method computeAndStoreNextPrime. You can even go ahead and use the "remove braces" intention on that while loop, as it now contains a single statement.
- This certainly made our generate method simple, only 2-3 lines. Of course now the work happens in computeAndStoreNextPrime.
- Let's take a look at the first if conditional in computeAndStoreNextPrime. It seems to be doing something with square: It's looking to see if the candidate prime is a square, and if it is then it computes the next prime square. In fact from the looks of it, that's the only place where square is being used, so replace the square in candidate == square with the value primes[ord] * primes[ord], and check your tests to make sure they still pass.
- If you scroll up now, you will notice that the declaration of square is grayed out. That's because now we don't *use* the value of square anywhere, we just set it. So go ahead and perform the "Remove field" intention.

Before we move on, we need to understand a bit more what this algorithm is doing. In order to check if a number is prime, it will have to check the multiples of primes less than it, but it does not need to worry about all such multiples: If a number is not a prime, then it will have a prime divisor that's no more than the square root of the number.

This is what the "square" check is all about: ord keeps track of the index of the next possible prime divisor: If a candidate is not prime then it will be a multiple of one of the

primes up to index ord. But when we reach the square of the prime that is at location ord, then this prime also becomes a possible factor and therefore we increment ord, and add this entry to the multiple array.

- With that in mind, let's extract the whole if conditional to a method called addNextMultipleEntryIfReachedNextPrimeSquare (if you can find a better name for it please share).
- Now go into that method, and notice the body of the if: It first increments ord, and then refers to the previous entry in the array. Let's merge the two: take the ord++ part and place it as the index instead of ord−1, and run your checks to make sure they still work. Now you should be able to eliminate the braces from the if conditional (intention).

Now let's go back to our computeAndStoreNextPrime method. The next thing that happens is that the index n starts at 2 and continues until ord, and for each of those indices we increment the multiple[n] entry until it is at least as big as the candidate. The multiple is incremented by adding two times the corresponding prime. What this effectively does is compute k∗p for odd numbers k only (since our target is always odd). This process stops if we discover that the candidate does equal one of these multiples, meaning that it is definitely not prime and we should move to the next candidate.

With a small cost in efficiency, we can separate those actions: We can increase the multiples to their next step, namely make them all reach at least the candidate, in one simple loop. Then we can have another loop that checks if any of those multiples equal the candidate. This will take some manual work:

- Right after the call to addNextMultipleEntryIfReachedNextPrimeSquare(); build a for loop for a variable n going from 2 up to ord by 1.
- In its body move the while (multiple[n] < candidate) { ... part (3 lines). Run your tests to make sure they still run.
- Use "Extract Method" to turn this body into a updateMultiplesToReachCandidate method.
- Now the remaining part, starting with the n=2 assignment and going just before the end of the big do-while loop, should be its own function. So extract it to a function called checkIfCandidateIsMultiple (we'll find a better name later).
- Thinking about it, the n field is now really only used in this one method, and it feels like it is just a looping variable. We should really have it as a local variable instead. Find the field declaration for n near the top of the file, and use the "Convert to local" intention. Make sure your tests still pass.
- Next let's think through this checkIfCandidateIsMultiple method. It seems to be updating this possiblyPrime field, which is used only in one place outside of the function, in the while (!possiblyPrime) entry. Why don't we simply have the function return a boolean instead, and make that actually return the opposite of possiblyPrime, namely candidateIsComposite. Let's see how we will do that:

  - Change the return time of checkIfCandidateIsMultiple to boolean.
  - At the bottom of the method, have it return !possiblyPrime. We'll improve on this later, but for now this will keep things working.
  - Find the place where checkIfCandidateIsMultiple is called, and perform "Extract Variable" on the call to store it to a value called candidateIsComposite. While we are at it, perform a "rename" on the function to have it called candidateIsComposite as well. Run your tests to check they pass.
  - Replace the while (!possiblyPrime) call to while (candidateIsComposite). You will see it all red, that's because the variable we created is declared inside the while scope. Perform the "bring boolean . . . into scope" intention. Make sure our tests still pass.
  - Now if we think about it for a second, the local variable we created isn't adding anything. So go ahead and perform the "Inline" refactoring on it. Check your tests again.

- Now it is time to simplify the candidateIsComposite method.
  - First of all, possiblyPrime is now only used within this method, so find the field declaration and perform the "Convert to local" intention.
  - Now it seems that instead of setting possiblyPrime to false, we could simply return true, so let's do that then check our tests.
  - Now you should see possiblyPrime light up. That's because the compiler has determined that possiblyPrime is going to always be true, and it offers to simplify the code for you. So go ahead and do the "Simplify possiblyPrime to true" intention in both places where it occurs. And you can further simplify the last return from !true to false. Finally, you can remove the possiblyPrime variable which is not used any more. Check the tests.
  - Now this while loop is clearly a for loop. Go ahead and rewrite it that way and check the tests again. You should now also be able to remove the braces from the if part.

Step 10: More cleanup

Let's do a bit more cleanup before wrapping up. Remember how we had split up the work of updating the multiples and then checking if they equal the candidate? Let's see if now that things are a bit more clear we can bring some of it back.

- In the updateMultiplesToReachCandidate method, perform "Extract Method" on the while loop to a method called updateNthMultiple. You can then remove the braces from the for loop.
- In candidateIsComposite, perform "Extract Method" on the multiple[n] == candidate part, to a method candidateIsNthMultiple (make sure to NOT fold the parameters).
- Thinking about it, we should perform the updateNthMultiple work inside candidateIsNthMultiple, so move it there from updateMultiplesToReachCandidate (leaving an empty loop in its place). Run your tests to make sure they still pass.

- Since updateMultiplesToReachCandidate(); now doesn't do anything, remove it from computeAndStoreNextPrime, and run your tests again. You can now also "Safe delete" the unused grayed-out method.
- Back in the computeAndStoreNextPrime method, perform "Extract Method" on the do–while loop to a method called computeNextPrime.
- Now it feels that this method should return this next prime, so have the method return candidate, and change its return time to int.
- Back in the computeAndStoreNextPrime method, use computeNextPrime(); instead of candidate in the assignment. Check your tests.
- Finally, instead of incrementing the index and then making the assignment, change the index in the assignment to ++primeIndex and eliminate the primeIndex++; line. This now turned into a nice one-liner!
- Now let's turn this computeNextPrime() call inside computeAndStoreNextPrime into a parameter using the "Extract Parameter" refactoring. Call the parameter nextPrime. Then rename the method to storeNextPrime.

Excellent, now let's order the methods according to the step down rule. Rearrange the methods in the following order:

- PrimeGenerator constructor
- generate
- needMorePrimes
- storeNextPrime
- computeNextPrime
- addNextMultipleEntryIfReachedNextPrimeSquare
- candidateIsComposite
- candidateIsNthMultiple
- updateNthMultiple

Step 11: From arrays to arraylists

Now that our code is nice and compartmentalized, let's see if we can use dynamic array lists instead of arrays, at least instead of the multiple array. The advantage is that we would not need the ordmax field at all, and we might be able to even eliminate the ord field. Since the primes array has a predetermined length anyway, we won't gain much from changing that (other than eliminating the primeIndex method perhaps).

In order to achieve this, the first step is to "encapsulate" access to the array elements behind getter and setter methods:

- Find the multiple[n] use in candidateIsNthMultiple, and "Extract Method" on it to a method called getNthMultiple (don't fold the parameters). Replace the one instance suggested.
- Find the multiple[ord++] = candidate; line in addNextMultipleEntryIfReachedNextPrimeSquare and "Extract Method" on it to a method addNewMultiple.
- In theory we would also need to do something about multiple[n] += primes[n] + primes[n];, but since it is the only occurence we will do our changes where it is.

Now we will perform our transformation steps in a specific order, in order to avoid breaking our tests. So we will do the following:

- Create a new field multipleList. We initialize it next to the multiple array.
- Whenever the multiple array gets updated, we *also* update the multipleList. Once that is done:
- Wherever the multiple array gets used, we change its usage to a usage of multipleList instead, and run our tests to make sure they did not break.
- Once the multiple array is not used any more, we eliminate it.
- We inline the getters and setters we created above.
- We will then look into the ord variable. This is related to how much of the array we are using, so it should be related to the size of the multipleList.

Ok so let's follow these steps one at a time:

- Right below the multiple = new int[ordmax + 1]; line that initialized the multiple array, type new ArrayList<Integer>() then "Extract Field" from it to a field called multipleList.
- Perform the "import class" intention to import the ArrayList module.
- Next, go to the addNewMultiple method, and add to it a line multipleList.add(candidate);.
- In the updateNthMultiple method, within the while loop, in addition to the current line, also add the lines int index = n − 2; and multipleList.set(index, multipleList.get(index) + primes[n] + primes[n]);. Check that our tests should still pass.
- Now we will start trying to use this class. In the getNthMultiple method, we want to return the "nth value" from the list. Since the multiple array started from the 2nd entry while our list starts from the 0th, we have an "off by 2" index factor to account. Comment out the current return value, and replace it with return multipleList.get(n − 2);. Make sure the tests pass, and then delete the commented out line.
- Now that the new list is used, let's start removing usages of the multiple array. Start with its use in updateNthMultiple, and run your tests.

- Then we need to do the same in addNewMultiple. Note however the ord++ part. As we are not ready to deal with that, replace the multiple[ord++] = candidate; mention with just ord++. Check that our tests still pass.
- Now the multiple field near the top should be grayed out, go ahead and remove it.
- You should now see ordmax also grayed out, so go ahead and remove that as well.
- Let's now rename our multipleList list to simply multiples.
- Next let's handle the issue of ord. Find a usage of ord, and "Extract Method" to getOrd(). Change all occurences.
- Now look at getOrd. Instead of it returning ord, change it to return multiples.size() + 2. Then make sure our tests still pass.
- Now see that ord is grayed out near the top of the file, and remove it and check our tests again.

Step 12: A class for multiples

If you look through the code now, you will find some weird indexing issues. Part of the reason is that when we think of how to update the value of a multiple, we need to look up in the primes array for the correct prime to use. And since the list and the primes array are indexed differently, it's a bit awkward.

What we will try to do here is avoid some of these problems by creating a small inner class called Multiple. It is meant to represent an entry in the multiples list, but it remembers the prime that needs to be added each time as well as its current value. The flip side to this is that we'll need to get the values in and out of that class. But let's try it out and see how it looks.

As before, we will do this by creating a new list to operate in parallel to the multiplesList. Then we'll gradually migrate from using one list to using the other.

- In the PrimeGenerator constructor add a call new ArrayList<Multiple>(); then "Extract Field" from it. Call it newMultiples.
- Then use the "Create inner class" intention on the red Multiple word to create this new inner class. Check that your tests still pass.
- The first place we need to change is where we add a new entry to the multiples list. In the addNewMultiple method, add the line newMultiples.add(new Multiple(primes[getOrd()]));. The idea is that will initialize the new object with the prime that it is responsible for.
  - Place your cursor on the red underline part, and perform the "Create constructor" intention. The parameter prime is well-named so leave it as is. However notice that it is grayed out. We must create a field out of it, so use the "Create field for parameter . . . " intention on it to create a field also named prime. Make sure it is set to be final.

  - We will also need a value field. Create it as follows: In the constructor, add the line prime * prime, then do "Extract Field" from that expression, set it in the Multiple class, and name the field value.
  - Back at addNewMultiple, it seems reasonable that primes[getOrd()] should be passed in as a parameter, so perform "Extract Parameter" to it to name the parameter prime.

- Next we need to find places where multiples elements change values, and do the analogous change to the Multiple objects we created. Look in updateNthMultiple at the multiples.set line. We need to add something similar for newMultiples, but the way that one will work will be to "get" the object and tell it to update its value.

  - Add newMultiples.get(index).updateValue(); to the body of the while loop there.
  - Now use the "Create method ..." intention to add a method in the Multiple class. The body of that method should say value += prime + prime; to perform the analogous update.

- Now we need to change the uses of the multiples class, which happen in the getNthMultiple method. Change that method to return newMultiples.get(n−2).getValue(); the use the "Create read-only property ..." intention on the getValue() expression to create a getValue method to the Multiple class which simply returns value. Run your tests to make sure they pass, because we really changed the system behavior here.

- Change the getOrd method to use newMultiples instead ofmultiples', and run your tests again.

- Now it is time to start removing the setting of values in multiples. Start with the multiples.set(...) line in updateNthMultiple, delete it and run the tests.

- Next, delete the multiples.add(...) from addNewMultiple, and run the tests.

- The multiples field declaration near the top should now be grayed out, so go ahead and do the "Remove field" intention on it then do a "Rename" refactoring on newMultiples to get them back to multiples.

Now we will reap the benefits of this: We should be able to gradually migrate from using n going from 2 to size+2 to using index going from 0 to size. We will employ a similar idea of gradually adding index handling next to the n handling until we can drop the n.

- We start in updateNthMultiple. Perform "Extract Parameter" on the index variable to turn it into an index parameter.

- Change the n in getNthMultiple(n) < candidate) to index + 2 and make sure your tests still pass. Then "Safe delete" the parameter n.

- Now in getNthMultiple, select the n−2 and do "Extract Parameter" to a parameter index. Notice that when you do this, the system automatically eliminated the other parameter. It also created the awkward index + 2 − 2 back in updateNthMultiple, so just simplify that to index.

- Now we move further up to candidateIsNthMultiple. Select the n−2 and "Extract Parameter" index, and tell it to replace all occurences. Run the tests again.
- Now look at candidateIsComposite. We will want to update that loop to a more normal loop. So manually replace it with a for loop that has an i variable going from 0 to multiples.size(), increasing by 1, then using i as an argument instead of n−2. The tests should still be passing.

Ok this is better, but looking at it more, is there really any reason to be passing in the index? Can't we be passing the elements themselves? Let's try that out:

- Start in the getNthMultiple method. Perform an "Extract Parameter" refactoring on the multiples.get(index) expression, to get a multiple parameter.
- In updateNthMultiple, again select multiples.get(index) and "Extract Parameter" for it, telling it to replace both occurences.
- You checked that your tests are still passing, right?
- Now let's do the same on candidateIsNthMultiple. Remember to tell it to replace all occurences.
- Now we have finally arrived candidateIsComposite, which now uses multiples.get(index) as an argument. Notice that the "for" loop has gray background to it. That means there is a simplification you can perform on it. Place your cursor on the for and use the "replace with foreach" intention.

There! Isn't this much better?

Well ok you are right, the methods below look a bit awkward, let's see what we can do about them.

- The getNthMultiple method is not really needed any more, just perform an "Inline" on it to eliminate it.
- It feels as though updateNthMultiple should really be part of the Multiple class. So place your cursor on it, and perform the "Move" refactoring.
- Hm its name is bad now, let's "Rename" it to updateToReach, then select primeGenerator.candidate and "Extract Parameter" on it to have a number parameter.
- Go ahead and "Inline" the getValue() and updateValue() usages (and remove them).
- Now the candidateIsNthMultiple method should probably be moved also, do a "Move" refactoring on it, followed by the "Extract Parameter" (replace both occurences) and "Rename" the method to becomes.

Take a look at our candidateIsComposite method now. Doesn't it look nice? Next up, let's tackle the issue of getOrd. It is now used for a single purpose, to get access to primes[getOrd()] in addNextMultipleEntryIfReachedNextPrimeSquare.

- Start by doing "Extract Method" on primes[getOrd()] to a method named nextPrimeFactor, replace all occurences.
- Now you can inline and remove the getOrd method.
- Next, select one of the nextPrimeFactor() calls, and "Extract Parameter" on it to get a parameter prime.
- Then let's go ahead and "Rename" the addNextMultipleEntryIfReachedNextPrimeSquare method to maybeAddNextMultiple, which is a lot shorter and reasonably descriptive.

As a final cleanup step, go to our PrimeGenerator constructor and perform the "Move assignment to field declaration" intention on all fields for which it is possible (all but primes and numberOfPrimes).

Also, check your method order for the Stepdown rule, and make sure it is followed.

Step 13: Transforming the primes array

Our last step in this long refactoring is the primes array. We would like to convert its use to an array list. Note that the first prime, 2, is placed at index 1 and also that the only primes used later in the algorithm (as part of the work on multiples) are the odd primes, which start at index 2. With that in mind, we will hold only the odd primes in our list. Then we will convert the result to an array when asked for.

- In the PrimeGenerator constructor, add a line that says new ArrayList<Integer>(); and then perform "Extract Field" on it to get a oddPrimes field. Then use the "Move assignment to declaration" intention to move the assignment out of the constructor.
- Now we need to find places where primes has values added to it, and also add those values to the oddPrimes list (skipping 2 of course). Find the storeNextPrime method, and add a oddPrimes.add(nextPrime); call to its body.
- Now we want to change access to the list to instead make use of this new list instead. Note that the indexing will be off by 2, which will work out well for us. Find the nextPrimeFactor method, and have it instead return oddPrimes.get(multiples.size());. Run your tests and notice that they fail! Something is going wrong here! So undo that last step, and let's think about it some more.

Notice the error we are getting: java.lang.IndexOutOfBoundsException: Index 0 out−of−bounds for length 0. This happens due to a call to nextPrimeFactor. That call is trying to get to the next available prime factor, but none has been created yet!

The solution is simple: We will initialize our list with the first odd prime, and also add it to the primes array, then start our candidate from 3 instead of 1.

- In the PrimeGenerator constructor, add lines primes[2] = 3; and oddPrimes.add(3);.
- In the declaration for candidate, initialize it at 3 rather than 1.

- Also primeIndex should start at 2 rather than 1. Run your tests to make sure they pass.
- Now try to replace the return in the method nextPrimeFactor with return oddPrimes.get(multiples.size()); and your tests should now pass.
- Now we need to gradually remove uses of primes. The biggest use is what we return from generate. Perform "Extract Method" on that return to form a method getPrimesArray instead. Don't change the other occurences here, we'll end up deleting them shortly.
- Now we need to implement getPrimesArray to instead build an array of primes from this oddPrimes list, together with the prime 2, and starting from index 1. First we need to initialize the array, which must have size 2 more than the list size. Then we do the appropriate copies, and finally return the array.

  ```
  int[] primes = new int[oddPrimes.size() + 2];
  primes[1] = 2;
  for (int i = 0; i < oddPrimes.size(); i++) {
    primes[i + 2] = oddPrimes.get(i);
  }
  return primes;
  ```

  Make sure your tests still pass.
- In storeNextPrime replace the line primes[++primeIndex] = nextPrime; with simply ++primeIndex. We'll fix that variable later, but this way we'll keep the effect there while removing the use of the primes array. Make sure your tests still pass.
- Now, primeIndex is still used in one place, namely in needMorePrimes. We should instead be using the oddPrimes list size, which should be 1 less than primeIndex. so replace primeIndex in that method with oddPrimes.size() + 1. Make sure your tests still pass.
- Now you should be able to eliminate the field primeIndex, and also remove the primes field. Check that your tests still pass.

And we're done! We could maybe do something more with the candidate += 2 bit, but this is more than good enough. The code should be easy to read at this point.