BASIC PROGRAMMING EXERCISES (CONDITIONAL STATEMENTS, LOOPS, METHODS, AND ARRAYS)

COMP-202A, Fall 2010, All Sections

INSTRUCTIONS

- Attempt each question on paper before trying to implement it in a Java program.
- Attempt to implement the solution to a problem only once you think you have a solution written on paper.
- Every loop **MUST** terminate as soon as it can; for example, if a question asks you to write a method which determines whether a given value occurs in a given array, the loop **MUST** terminate as soon as it finds a value in the given array which is equal to the given value, without looking at the rest of the values. However, you **MUST NOT** use the `break` statement to exit loops.
- Each question asks you to write a class which defines one method. After completing each question, add a `main()` method to the class you wrote for that question. This `main()` method should ask the user to enter values for each of the parameters of the method you wrote for that question, and read these values from the keyboard. Then, have your `main()` method call the method you wrote for that question, pass the values the user entered as parameters to this method, and display the results this method returns.
- To read an array of `char` from the keyboard, use

  ```
  keyboard.nextLine().toCharArray();
  ```

  where `keyboard` is the name of the `Scanner` variable you initialized to read from the keyboard. Likewise, to read a single `char` from the keyboard, use

  ```
  keyboard.nextLine().charAt(0);
  ```

  where, again, `keyboard` is the name of the `Scanner` variable you initialized to read from the keyboard.

PROBLEMS

1. Write a Java class called `Factorial`. This class defines a method called `factorial()` which takes as its only parameter an `int` called `n`, and returns an `int` representing the factorial of `n`. The factorial of an integer n, denoted $n!$, is defined as $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (but note that 0! is 1). You **MAY** assume that `n` is a non-negative integer.
2. Write a Java class called `Fibonacci`. This class defines a method called `fibonacci()` which takes as its only parameter an `int` called `n`, and returns an `int` representing the $\texttt{n}^{\text{th}}$ Fibonacci number. The

n^{th} Fibonacci number, denoted f_n, is defined as follows:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

You **MAY** assume that `n` is a non-negative integer.

3. Write a Java class called `PrimalityChecker`. This class defines a method called `isPrime()`, which takes as its only parameter an `int` called `n`, and returns a `boolean` which is `true` if and only if `n` is a prime number, `false` otherwise. A prime number is an integer which cannot be divided evenly by any integer except 1 and itself. You **MAY** assume that `n` is a non-negative integer.

4. Write a Java class called `GoldbackChecker`. This class defines a method called `checkGoldbach()`, which takes as its only parameter an `int` called `n`, and returns an array of `int`. This method finds two prime numbers whose sum is equal to `n`, and returns these two prime numbers in an array of `int`s of length `2`; if two such primes cannot be found, `n` is less than 4, or `n` is odd, your method should return `null`. Your `checkGoldbach()` method **MAY** call the `isPrime()` method you wrote for a previous exercise. Note that every even `int` value greater than `4` can be expressed as the sum of two prime numbers.

5. Write a Java class called `Power`. This class defines a method called `power()` which takes as parameters a `double` called `base` as well as an `int` called `exponent`, and returns a `double` representing the value of `base` raised to the power `exponent`. You **MAY** assume that `exponent` is a non-negative integer value, and that `base` and `exponent` are not both `0`. You **MUST** write the computation yourself; in other words, you **MUST NOT** use the `Math.pow()` method.

6. Write a Java class called `CountDigits`. This class defines a method called `countDigits()` which takes as its only paramter an `int` called `n`, and returns an `int` representing the number of digits in `n`. You **MAY** assume that `n` is a positive integer.

7. Write a Java class called `Contains`. This class defines a method called `contains()`, which takes as parameters an array of `int` called `a` as well as an `int` called `x`, and returns a `boolean` which is `true` if and only if `x` occurs in `a`, `false` otherwise.

8. Write a Java class called `ContainsInRange`. This class defines a method called `contains()`, which takes as parameters an array of `int` called `a`, an `int` called `x`, an `int` called `start`, as well as an `int` called `end`, and returns a `boolean` which is `true` if and only if `x` occurs in `a` at a position which is greater than or equal to `start`, and less than `end`. You **MAY** assume that `start` is less than or equal to `end`, and that both `start` and `end` are greater than or equal to 0, and less than or equal to `a.length`.

9. Write a Java class called `CheckDuplicates`. This class defines a method called `checkDuplicates()`, which takes as its only parameter an array of `int` called `a`, and returns a `boolean` which is `true` if and only if there exists at least one value in `a` which occurs more than once. Your `checkDuplicates()` method **MAY** call the `contains()` method you wrote for a previous exercise.

10. Write a Java class called `Maximum`. This class defines a method called `maximum()`, which takes as its only parameter an array of `int` called `a`, and returns an `int` representing the maximum value which occurs in the array.

11. Write a Java class called `CountOccurrences`. This class defines a method called `countOccurrences()`, which takes as parameters an array of `int` called `a` as well as a `int` called `x`, and returns an `int` representing the number of elements of `a` which are equal to `x`.

12. Write a Java class called `MostOftenOccurring`. This class defines a method called `occursMostOften()`, which takes as its only parameter an array of `int` called `a` and returns an `int` representing the value which occurs most often in `a`. If there is a tie, display the value which occurs first in `a` among those that occur most often. Your `occursMostOften()` method **MAY** call the `countOccurrences()` method you wrote for the previous exercise.

13. Write a Java class called `ToUpperCase`. This class defines a method called `toUpperCase()`, which takes as its only parameter an array of `char` called `s`, and returns an array of `char`. The array that the method returns contains exactly the same characters as `s` in the same order, except that all lower-case letters occurring in `s` are replaced by their upper-case equivalents in the array returned by the method.

14. Write a Java class called `OccursHere`. This class defines a method called `occursHere()`, which takes as parameters two arrays of `char` called `superString` and `subString`, as well as a positive integer `position`, and returns a `boolean`. This `boolean` is `true` if and only if all the characters in `subString` occur consecutively in `superString` starting at position `position`, in the same order as the one in which they appear in `subString`, `false` otherwise.

15. Write a Java class called `IndexOf`. This class defines a method called `indexOf()`, which takes as parameters two arrays of `char` called `superString` and `subString`, and returns an `int`. This `int` represents the smallest index within `superString` at which the characters of `subString` appear consecutively in `superString`, in the same order as the one in which they appear in `subString`. This method returns `-1` if the characters of `subString` never appear consecutively in `superString`. Your `indexOf()` method **MAY** call the `occursHere()` method you wrote for a previous exercise.

16. Write a Java class called `Substring`. This class defines a method called `substring()`, which takes as parameters an array of `char` called `s`, an `int` called `beginIndex`, as well as an `int` called `endIndex`, and returns a new array of `char` which consists only of the characters of `s` between positions `beginIndex` (inclusive) and `endIndex` (exclusive). You **MAY** assume that `beginIndex` is greater than or equal to 0, that `endIndex` is less than or equal to the length of `s`, and that `beginIndex` is less than or equal to `endIndex`.

17. Write a Java class called `RemoveAll`. This class defines a method called `removeAll()`, which takes as parameters an array of `int` called `a` as well as an `int` called `x`, and returns a new array of `int` which consists of all the elements in `a`, in the order in which they appear in `a`, but with all occurrences of `x` removed. The size of the array returned by this method **MUST** be equal to the number of elements it contains; that is, it must be equal to `a.length - n`, where `n` is the number of occurrences of `x` in `a`. Your `removeAll()` method **MAY** call the `countOccurrences()` method you wrote for a previous exercise.

18. Write a Java class called `NumberConsecutive`. This class defines a method called `countOccurrences()` that takes as parameters an array of `int` called `a`, an `int` called `x`, as well as an `int` called `position`, and returns an `int` representing the number of occurrences of `x` in `a` starting at position `position`. For example, if `a` is {`1, 2, 2, 2, 2, 2, 1, 1, 1`}, `x` is `2`, and `position` is `1`, then your method will return `5`, as there are 5 consecutive occurrences of `2` in `a` starting at position `1`. You **MAY** assume that `position` is greater than or equal to `0` and less than `a.length`

19. Write a Java class called `MaxConsecutive`. This class defines a method called `maxOccurrences()` that takes as parameters an array of `int` called `a` as well as an `int` called `x`, and returns an `int` representing the position in `a` at which the longest series of consecutive occurrences of `x` occurs. If `x` never occurs in `a`, then the method should return `-1`. If there is a tie between two positions, you should return the smallest position. Your `maxOccurrences()` method **MAY** call the `countOccurrences()` method of the `NumberConsecutive` class you wrote for a previous exercise.

20. Write a Java class called `Reverse`. This class defines a method called `reverse()`, which takes as its only parameter an array of `int` called `a`, and returns a new array of `int` which contains all the elements in `a`, but in the reverse order.

21. Write a Java class called `ReverseInPlace`. This class defines a method called `reverse()`, which takes as its only parameter an array of `int` called `a`, and returns `void`. This method changes `a` so that when the method returns, its elements occur in the reverse order.

22. Write a Java class called `Palindrome`. This class defines a method called `isPalindrome()`, which takes as its only parameter an array of `char` called `s`, and returns a `boolean` which is `true` if and only if the characters in `s` form a palindrome. A palindrome is a series of characters which reads the same

forwards and backwards, such as `"laval"` or `"stressed desserts"`.

23. Write a Java class called `Equals`. This class defines a method called `equals()`, which takes as parameters two arrays of `int` called `a1` and `a2`, and returns a `boolean` which is `true` if and only if `a1` and `a2` are equal. For the purposes of this question, two arrays of `int` are equal if they are of the same length and the same values appear in the same positions.

24. Write a Java class called `Subset`. This class defines a method called `isSubset()`, which takes as parameters two arrays of `int` called `big` and `small`, and returns a `boolean` which is `true` if and only if all the elements in `small` also occur in `big`. Your `isSubset()` method **MAY** call the `contains()` method you wrote for a previous exercise.

25. Write a Java class called `CountWords`. This class defines a methods called `countWords()`, which takes as its only parameter an array of `char` called `s`, and returns an `int` representing the words in `s`. A word is delimited by one or many space characters.

MORE PROBLEMS

The Java standard class library provides a variety of classes. Some of these classes, such as the `Math` and `Arrays` class, provide various class methods which perform common tasks. The following exercises consist of of writing classes that provide methods which are similar to those provided by the `Math` and `Arrays` classes, and will allow you to gain some insight on how these two classes work, in addition to provided you with good practice for the midterm.

Note the following:

- Before you begin this exercise, experiment with the methods provided by the Math class and Arrays class. How does one use them in a program? What do they do? This information is available at `http://download.oracle.com/javase/6/docs/api/`
- The methods you write for this exercise **SHOULD** call other methods you have written for the exercise whenever doing so is useful.
- Finally, you **MUST NOT** call any method defined in the `Math` or `Arrays` classes to complete this exercise.

Reimplementing the `Math` class

The `Math` class, which is part of the Java standard class library, provides various class methods that perform common calculations, such as trigonometric functions.

Write a `public` class called `MyMath`, which provides some of the functionality which is also provided by the `Math` class. This class should define the `public` and `static` methods described below:

1. A method called `toDegrees()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an angle expressed in radians, and the value returned by the method represents the same angle, but expressed in degrees (360 degrees = 2π radians). For example, suppose the following method call is executed:

   ```
   double degrees = MyMath.toDegrees(3.141592653589793);
   ```

   After the execution of the above method call, the value stored in variable `degrees` will be (approximately) `180.0`, as 3.141592653589793 radians is equivalent to 180 degrees. You **MUST** use `3.141592653589793` as a value for π.

2. A method called `toRadians()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an angle expressed in degrees, and the value returned

by the method represents the same angle, but expressed in radians (again, 360 degrees = 2π radians). For example, suppose the following method call is executed:

```
double radians = MyMath.toRadians(180.0);
```

After the execution of the above method call, the value stored in variable `radians` will be (approximately) `3.141592653589793`, as 180 degrees is equivalent to 3.141592653589793 radians. Again, you **MUST** use `3.141592653589793` as a value for π.

3. A method called `absoluteValue()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an arbitrary number, and the value returned by the method represents the absolute value of this arbitrary number. The absolute value of a number x, denoted $|x|$, is x if x is greater than or equal to 0, and $-x$ if x is less than 0. For example, suppose the following method call is executed:

   ```
   double value = MyMath.absoluteValue(-42.0);
   ```

   After the execution of the above method call, the value stored in variable `value` will be `42.0`, as the absolute value of -42 is 42.

4. A method called `minimum()`, which takes as parameters two values of type `double`, and returns a value of type `double`. The parameters represent two arbitrary numbers, and the value returned by this method represents the smaller of these two arbitrary numbers; if the two numbers are equal, then the method may return either of the two. For example, suppose the following method call is executed:

   ```
   double min = MyMath.minimum(1.0, 2.0);
   ```

   After the execution of the above method call, the value stored in variable `min` will be `1.0`, as 1 is less than 2.

5. A method called `maximum()`, which takes as parameters two values of type `double`, and returns a value of type `double`. The parameters represent two arbitrary numbers, and the value returned by this method represents the larger of these two arbitrary numbers; if the two numbers are equal, then the method may return either of the two. For example, suppose the following method call is executed:

   ```
   double max = MyMath.maximum(1.0, 2.0);
   ```

   After the execution of the above method call, the value stored in variable `min` will be `2.0`, as 2 is greater than 1.

6. A method called `power()`, which takes as parameters a value of type `double` followed by a value of type `int`, and returns a value of type `double`. The `double` parameter represents an arbitrary base a, the `int` parameter represents a non-negative exponent b, and the value returned by the method represents a^b. For example, suppose the following method call is executed:

   ```
   double x = MyMath.power(1.5, 4);
   ```

   After the execution of the above method call, the value stored in variable `x` will be `5.0625`, as $1.5^4 = 5.0625$. You **MAY** assume that the value of the `int` parameter is greater than or equal to `0`; in other words, you do not have to handle cases where the value of the `int` parameter is negative.

7. A method called `root()`, which takes as parameters a value of type `double` followed by a value of type `int`, and returns a value of type `double`. The `double` parameter represents an arbitrary number n, the `int` parameter represents a positive root r, and the value returned by the method represents the r^{th} root of n, denoted $\sqrt[r]{n}$. For example, suppose the following method call is executed:

   ```
   double x = MyMath.root(5.0625, 4);
   ```

   After the execution of the above method call, the value stored in variable `x` will be `1.5`, as $\sqrt[4]{5.0625} = 1.5$ ($1.5^4 = 5.0625$).

   To calculate the r^{th} root of n, where r is an integer and n is a real number, you can use the following algorithm:

- Start with a guess g of 1.
- Calculate g' using the following formula:

$$g' = g - \frac{g^r - n}{rg^{r-1}}$$

If $|g' - g| < 10^{-10}$, then $g' \approx \sqrt[r]{n}$. Otherwise, set the new value of g to be the current value of g' and repeat this step.

You **MAY** assume that the value of the `int` parameter is greater than or equal to `1`, and that the `double` parameter is greater than or equal to `0.0`; in other words, you do not have to handle cases where the value of the `int` parameter is negative or `0`, or cases where the value of the `double` parameter is negative.

8. A method called `gcd()`, which takes as parameters two values of type `int`, and returns a value of type `int`. The `int` parameters represent arbitrary positive integers, and the value returned by the method represents the greatest common divisor of the two `int` parameters (that is, the largest positive integer which is a divisor of both `int` parameters). For example, suppose the following method call is executed:

```
double divisor = MyMath.gcd(24, 18);
```

After the execution of the above method call, the value stored in variable `divisor` will be `6`, as 6 is the largest positive integer which divides both 24 and 18 without remainder.

The greatest common divisor of two positive integers a and b, denoted $\gcd(a, b)$, can be computed using the following algorithm:

- If $b = 0$, then $\gcd(a, b) = a$; conversely, if $a = 0$, then $\gcd(a, b) = b$.
- Otherwise, $\gcd(a, b) = \gcd(b, a \bmod b)$

You **MAY** assume that the values of both `int` parameters are positive; in other words, you do not have to handle cases where the value either `int` parameter is negative.

9. A method called `lcm()`, which takes as parameters two values of type `int`, and returns a value of type `int`. The `int` parameters represent arbitrary positive integers, and the value returned by the method represents the least common multiple of the two `int` parameters (that is, the smallest positive integer which is a multiple of both `int` parameters). For example, suppose the following method call is executed:

```
double multiple = MyMath.lcm(4, 6);
```

After the execution of the above method call, the value stored in variable `multiple` will be `12`, as 12 is the smallest positive integer which is a multiple of both 4 and 6.

The least common multiple of two positive integers a and b, denoted $\text{lcm}(a, b)$, can be calculated using the following formula:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

You **MAY** assume that the values of both `int` parameters are positive; in other words, you do not have to handle cases where the value either `int` parameter is negative.

10. A method called `sine()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an angle expressed in radians, and the value returned by the method represents the trigonometric sine of this angle. For example, suppose the following method call is executed:

```
double sin = MyMath.sine(1.047197551);
```

After the execution of the above method call, the value stored in variable `sin` will be (approximately) `0.866025404`, as the sine of 1.047197551 radians is approximately 0.866025404.

The sine of an angle x, denoted $\sin x$, can be calculated using the following algorithm:

- If $x < -\pi$, repeatedly add 2π to x until $-\pi \leq x \leq \pi$. Conversely, if $x > \pi$, repeatedly subtract 2π from x until $-\pi \leq x \leq \pi$. You **MUST** use `3.141592653589793` as a value for π.
- Calculate $\sin x$ using the following formula:

$$\begin{aligned}\sin x &= \sum_{i=0}^{n} t_i \\ &= t_0 + t_1 + t_2 + \ldots + t_n\end{aligned}$$

where $t_0 = x$. To calculate the other terms in the sum, we use the following formula:

$$t_{i+1} = t_i \cdot \frac{-x^2}{(2i+3)(2i+2)}$$

where t_i is an arbitrary term, and t_{i+1} is the next term. For example:

$$t_1 = t_0 \cdot \frac{-x^2}{(2 \cdot 0 + 3)(2 \cdot 0 + 2)}$$
$$t_2 = t_1 \cdot \frac{-x^2}{(2 \cdot 1 + 3)(2 \cdot 1 + 2)}$$
$$t_3 = t_2 \cdot \frac{-x^2}{(2 \cdot 2 + 3)(2 \cdot 2 + 2)}$$

and so on. The last term in this sum, t_n, will be the first term whose absolute value is less than 10^{-10}.

11. A method called `cosine()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an angle expressed in radians, and the value returned by the method represents the trigonometric cosine of this angle. For example, suppose the following method call is executed:

```
double cos = MyMath.cosine(1.047197551);
```

After the execution of the above method call, the value stored in variable `cos` will be (approximately) `0.5`, as the cosine of 1.047197551 radians is approximately 0.5.

The cosine of an angle x, denoted $\cos x$, can be calculated using the same algorithm as the one used to calculate the sine of an angle x, except that when calculating the $\cos x$, $t_0 = 1$ and the formula to calculate the next term from the current term is:

$$t_{i+1} = t_i \cdot \frac{-x^2}{(2i+2)(2i+1)}$$

12. A method called `tangent()`, which takes as its only parameter a value of type `double`, and returns a value of type `double`. The parameter represents an angle expressed in radians, and the value returned by the method represents the trigonometric tangent of this angle. For example, suppose the following method call is executed:

```
double tan = MyMath.tangent(1.047197551);
```

After the execution of the above method call, the value stored in variable `tan` will be (approximately) `1.732050807`, as the tangent of 1.047197551 radians is approximately 1.732050807.

The tangent of an angle x, denoted $\tan x$, can be calculated using the following formula:

$$\tan x = \sin x / \cos x$$

Reimplementing the `Arrays` class

The `Arrays` class, which is also part of the Java standard class library, provides various class methods that perform common tasks on arrays, such as searching and sorting.

Write a `public` class called `MyArrays`, which provides some of the functionality which is also provided by the `Arrays` class. This class should define the `public` and `static` methods described below:

1. A method called `linearSearch()`, which takes as parameters an array of `int` followed by three values of type `int`, and returns a value of type `int`. The first `int` parameter represents a key, the second `int` parameter represents a starting position, and the third `int` parameter represents an end position. If the key occurs in the array between the start position (inclusive) and the end position (exclusive), the method returns the position of the first occurrence of the key in that range; otherwise, the method returns `-1`. For example, suppose the following method call is executed:

   ```
   int[] myArray = {4, 5, 6, 5, 7, 1, 5, 9};
   int position = MyArrays.linearSearch(myArray, 5, 2, 7);
   ```

   After the execution of the above method call, the value stored in variable `position` will be `3`, as the first occurrence of `5` in `myArray` between positions `2` (inclusive) and `7` (exclusive) is at position `3`.

   The values stored in the array can appear in any order. You **MAY** assume that the parameter values satisfy all of the following preconditions:

   - The value of the starting position is greater than or equal to `0`, and less than the size of the array.
   - The value of the end position is greater than or equal to `0`, and less than or equal to the size of the array.
   - The value of the start position is less than or equal to the value of the end position.
   - The array parameter is not a `null` reference.

   In other words, your method does not have to handle cases where one or more of the above conditions are not satisfied.

2. A method called `linearSearch()`, which takes as parameters an array of `int` followed by a single value of type `int`, and returns a value of type `int`. The single `int` parameter represents a key; if this key occurs anywhere in the array, the method returns the position of the first occurrence of the key in the array; otherwise, the method returns `-1`. For example, suppose the following method call is executed:

   ```
   int[] myArray = {4, 5, 6, 5, 7, 1, 5, 9};
   int position = MyArrays.linearSearch(myArray, 5);
   ```

   After the execution of the above method call, the value stored in variable `position` will be `1`, as the first occurrence of `5` in the array `myArray` is at position `1`.

   The values stored in the array can appear in any order. You **MAY** assume that the array parameter is not a `null` reference; in other words, your method does not have to handle cases where the above condition is not satisfied.

3. A method called `equals()`, which takes as parameters two arrays of `int` and returns a value of type `boolean`. This `boolean` value is `true` if the two parameter arrays are equal, `false` otherwise. Two arrays are equal if and only if they contain the same elements in the same order; that is, in order for two arrays to be equal, they must contain the same number of elements, and for every valid position i in these arrays, the value stored at position i in the first array must be equal to the value stored at position i in the second array. Note that if both arrays are `null` references, they are also considered equal.

   For example, suppose the following method call is executed:

```
int[] left = {1, 3, 3};
int[] right = {1, 3, 3};
boolean same = MyArrays.equals(left, right);
```

After the execution of the above method call, the value stored in variable `same` will be `true`, as arrays `left` and `right` contain the same values in the same order. Your method **MUST** correctly handle the case where one of the two parameter arrays is a `null` reference, but not the other.

4. A method called `fill()`, which takes as parameters an array of `int` followed by three values of type `int`, and returns nothing. The first parameter represents an arbitrary value, the second `int` parameter represents a starting position, and the third `int` parameter represents an end position. The method then assigns a value equal to the first `int` parameter to each element of the array whose position is between the starting position (inclusive) and the end position (exclusive).

   For example, suppose the following method call is executed:

   ```
   int[] myArray = {2, 3, 5, 7, 11, 13, 17, 19};
   MyArrays.fill(myArray, 0, 2, 6);
   ```

   After the execution of the above method call, the contents of the array whose address is stored in variable `myArray` will be {`2, 3, 0, 0, 0, 0, 17, 19`}, as every element whose position is between `2` (inclusive) and `6` (exclusive) is replaced with a `0`.

   Note that if the starting position is equal to the end position, then none of the values stored in the array are changed.

   You **MAY** assume that the parameter values satisfy all of the following preconditions:

   - The value of the starting position is greater than or equal to `0`, and less than the size of the array.
   - The value of the end position is greater than or equal to `0`, and less than or equal to the size of the array.
   - The value of the start position is less than or equal to the value of the end position.
   - The array parameter is not a `null` reference.

   In other words, your method does not have to handle cases where one or more of the above conditions are not satisfied.

5. A method called `fill()`, which takes as parameters an array of `int` followed by a single value of type `int`, and returns nothing. The method then assigns a value equal to the first `int` parameter to each element in the array.

   For example, suppose the following method call is executed:

   ```
   int[] myArray = {2, 3, 5, 7, 11, 13, 17, 19};
   MyArrays.fill(myArray, 0);
   ```

   After the execution of the above method call, the contents of the array whose address is stored in variable `myArray` will be {`0, 0, 0, 0, 0, 0, 0, 0`}, as every element in the array is replaced with a `0`.

   You **MAY** assume that the array parameter is not a `null` reference; in other words, your method does not have to handle cases where the above condition is not satisfied.

6. A method called `copyOf()`, which takes as parameters an array of `int` followed by two values of type `int`, and returns an array of `int`. The first `int` parameter represents a starting position, and the third `int` parameter represents an end position. This method creates a new array, and copies all elements of the parameter array whose positions are between the starting index (inclusive) and the end index (exclusive) so that the order in which the values appear in the new array is the same as the order in which they appear in the parameter array.

For example, suppose the following method call is executed:

```
int[] myArray = {2, 3, 5, 7, 11, 13, 17, 19};
int[] copy = MyArrays.copyOf(myArray, 2, 6);
```

After the execution of the above method call, the contents of the array whose address is stored in variable `copy` will be {`5, 7, 11, 13`}, as every element in `myArray` whose position is between `2` (inclusive) and `6` (exclusive) is copied to the new array.

If the starting position is equal to the end position, then none of the values stored in the parameter array are copied. The size of the array returned by this method is exactly equal to the number of elements copied from the parameter array.

You **MAY** assume that the parameter values satisfy all of the following preconditions:

- The value of the starting position is greater than or equal to `0`, and less than the size of the array.
- The value of the end position is greater than or equal to `0`, and less than or equal to the size of the array.
- The value of the start position is less than or equal to the value of the end position.
- The array parameter is not a `null` reference.

In other words, your method does not have to handle cases where one or more of the above conditions are not satisfied.

7. A method called `copyOf()`, which takes as its only parameter an array of `int`, and returns an array of `int`. This method creates a new array whose length is the same as the parameter array, and copies all elements of the parameter array into the new array, so that the order in which the values appear in the new array is the same as the order in which they appear in the parameter array.

   For example, suppose the following method call is executed:

   ```
   int[] myArray = {2, 3, 5, 7, 11, 13, 17, 19};
   int[] copy = MyArrays.copyOf(myArray);
   ```

   After the execution of the above method call, the contents of the array whose address is stored in variable `copy` will be {`2, 3, 5, 7, 11, 13, 15, 19`}, as every element of the original array is copied to the new array.

   Subsequent changes to the elements stored in the array returned by this method **MUST NOT** affect the parameter array; conversely, subsequent changes to the elements stored in the parameter array **MUST NOT** affect the array returned by this method.

   You **MAY** assume that the array parameter is not a `null` reference; in other words, your method does not have to handle cases where the above condition is not satisfied.

8. A method called `sort()`, which takes as parameters an array of `int` followed by two values of type `int`, and returns nothing. The first `int` parameter represents a starting position, and the third `int` parameter represents an end position. The method modifies the array so that the values occurring between the start position (inclusive) and the end position (exclusive) appear in increasing order.

   There are multiple ways to sort the elements of an array. However, for the purposes of this assignment, you **MUST** implement the following simple sorting algorithm:

   - Set i to be the starting position
   - Find the minimum element occurring at a position between i (inclusive) and the end position (exclusive), and swap this element with the one at position i.
   - If i is equal to the end position minus 1, then stop; otherwise, increment i and go back to the previous step.

For example, suppose the following method call is executed:

```
int[] myArray = {6, 2, 5, 7, 1, 8, 4, 3};
MyArrays.sort(myArray, 2, 6);
```

After the execution of the above method call, the contents of the array whose address is stored in variable `myArray` will be {`6, 2, 1, 5, 7, 8, 4, 3`}, as every element in `myArray` whose position is between `2` (inclusive) and `6` (exclusive) is sorted in increasing order.

You **MAY** assume that the parameter values satisfy all of the following preconditions:

- The value of the starting position is greater than or equal to `0`, and less than the size of the array.
- The value of the end position is greater than or equal to `0`, and less than or equal to the size of the array.
- The value of the start position is less than or equal to the value of the end position.
- The array parameter is not a `null` reference.

In other words, your method does not have to handle cases where one or more of the above conditions are not satisfied.

9. A method called `sort()`, which takes as its only parameter an array of `int`, and returns nothing. The method modifies the array so that all the values it contains appear in increasing order. This method **MUST** implement the same sorting algorithm as the one implemented by the other `sort()` method you wrote.

   For example, suppose the following method call is executed:

   ```
   int[] myArray = {6, 2, 5, 7, 1, 8, 4, 3};
   MyArrays.sort(myArray);
   ```

   After the execution of the above method call, the contents of the array whose address is stored in variable `myArray` will be {`1, 2, 3, 4, 5, 6, 7, 8`}, as every element in `myArray` is sorted in increasing order.

   You **MAY** assume that the array parameter is not a `null` reference; in other words, your method does not have to handle cases where the above condition is not satisfied.

10. A method called `toString()`, which takes as its only parameter an array of `int`, and returns a single `String`. This method produces a textual representation of the contents of the parameter array. This textual representation consists of the concatenation of the following elements:

    - The `String "["`
    - The text representation of each element in the parameter array; each pair of adjacent elements is separated by the `String ", "`
    - The `String "]"`

    In the textual representation of the parameter array, the array elements occur in the same order as in the parameter array.

    For example, suppose the following method call is executed:

    ```
    int[] myArray = {6, 2, 5, 7, 1, 8, 4, 3};
    String text = MyArrays.toString(myArray);
    ```

    After the execution of the above method call, the `String` stored in in variable `text` will be `"[6, 2, 5, 7, 1, 8, 4, 3]"`.

    If the array parameter is a `null` reference, your method **MUST** return the `String "null"`.