ARRAY-1 BASICS

```
int[] nums = { 6, 28, 34, ... 109, -87, -15 };
int lenN = nums.length;
int first = nums[0];  // 6
int second = nums[1]; // 28
int third = nums[2]; //34
int last = nums[lenN-1]; // -15
int secondToLast = nums[lenN-2]; // -87
int thirdToLast = nums[lenN-3]; // 109
```

Position	1st	2nd	3rd	…	3rd to Last	2nd to Last	Last
nums	6	28	34		109	-87	-15
Variable	nums[0]	nums[1]	nums[2]		nums[lenN-3]	nums[lenN-2]	nums[lenN-1]
Index (position)	0	1	2		len-3	len-2	len-1

```
nums[0] = 3; // changes the value of the 1st element to 3
nums[lenN-1] = 2; // changes the value of the last element to 2
```

Position	1st	2nd	3rd	…	3rd to Last	2nd to Last	Last
nums	3	28	34		109	-87	2
Variable	nums[0]	nums[1]	nums[2]		nums[lenN-3]	nums[lenN-2]	nums[lenN-1]
Index (position)	0	1	2		len-3	len-2	len-1

```
String[] txt = { "We", "the", "people", ... "a", "more", "perfect" };
int lenT = txt.length;
int first = txt[0]; // "We"
int second = txt[1]; // "the"
int third = txt[2]; // "people"
int last = txt[lenT-1]; // "perfect"
int secondToLast = txt[lenT-2]; // "more"
int thirdToLast = txt[lenT-3]; // "a"
```

Position	1st	2nd	3rd	…	3rd to Last	2nd to Last	Last
txt	"We"	"the"	"people"		"a"	"more"	"perfect"
Variable	txt[0]	txt[1]	txt[2]		txt[lenT-3]	txt[lenT-2]	txt[lenT-1]
Index (position)	0	1	2		len-3	len-2	len-1

```
txt[1] = "those"; // changes the value of the 2nd-element to "those"
txt[lenT-2] = "better"; // changes the value of the 2nd-to-last element to "better"
```

Position	1st	2nd	3rd	…	3rd to Last	2nd to Last	Last
txt	"We"	"those"	"people"		"a"	"better"	"perfect"
Variable	txt[0]	txt[1]	txt[2]		txt[lenT-3]	txt[lenT-2]	txt[lenT-1]
Index (position)	0	1	2		len-3	len-2	len-1

ARRAY-1 BASICS

EVEN-length Array (no single middle element, but rather 2 middle elements)

```
int[] nums = { 8, 18, 28, -5, 56, 35 };
int lenN = nums.length; // 6
int midN = lenN / 2; // 6 ÷ 2 = 3
int firstMiddle = nums[midN-1]; // 28  (Last element of the 1st half)
int secondMiddle = nums[midN]; // -5   (1st element of the 2nd half)
```

The element at position `midN-1` is the **Last element of the 1st half**.
The element at position `midN` is the **1st element of the 2nd half**.

Position	1st	2nd	3rd	4th	5th	6th
nums	8	18	28	-5	56	35
	nums[0]	nums[1]	nums[midN-1]	nums[midN]	nums[4]	nums[5]
Index (position)	0	1	2 mid-1	3 mid	4	5

```
nums[midN-1] = 50; // change the value of the 1st Middle element to 50
nums[midN] = 100; // change the value of the 2nd Middle element to 100
```

Position	1st	2nd	3rd	4th	5th	6th
nums	8	18	50	100	56	35
	nums[0]	nums[1]	nums[midN-1]	nums[midN]	nums[4]	nums[5]

```
String[] txt = { "We", "the", "people", "of", "the", "U.S." };
int lenT = txt.length; // 6
int midT = lenT / 2; // 6 ÷ 2 = 3
int firstMiddle = txt[midT-1]; // "people"; (Last element of the 1st half)
int secondMiddle = txt[midT]; // "of"  (1st element of the 2nd half)
```

The element at position `midT-1` is the **Last element of the 1st half**.
The element at position `midT` is the **1st element of the 2nd half**.

Position	1st	2nd	3rd	4th	5th	6th
txt	"We"	"the"	"people"	"of"	"the"	"U.S."
	txt[0]	txt[1]	txt[midT-1]	txt[midT]	txt[4]	txt[5]
Index (position)	0	1	2 mid-1	3 mid	4	5

```
txt[midT-1] = "humans"; // change the value of the 1st Middle element to "humans"
txt[midT] = "inside"; // change the value of the 2nd Middle element to "inside"
```

Position	1st	2nd	3rd	4th	5th	6th
txt	"We"	"the"	"humans"	"inside"	"the"	"U.S."
	txt[0]	txt[1]	txt[midT-1]	txt[midT]	txt[4]	txt[5]

ARRAY-1 BASICS

ODD-length Array (has a true middle element)

```
int[] nums = { 8, 18, 28, -5, 56, 35, 89 };
int lenN = nums.length; // 7
int midN = lenN / 2; // 7 ÷ 2 = 3 (integer division! Throw away the fraction!)
int middle = nums[midN]; // -5  (TRUE middle element)
int beforeMiddle = nums[midN-1]; // 28  (the element BEFORE the middle element)
int afterMiddle = nums[midN+1]; // 56  (the element AFTER the middle element)
```

Position	1st	2nd	3rd	4th	5th	6th	7th
nums	8	18	28	-5	56	35	89
	nums[0]	nums[1]	nums[midN-1]	nums[midN]	nums[midN+1]	nums[5]	nums[6]
Index (position)	0	1	2 mid-1	3 mid	4 mid+1	5	6

```
nums[midN-1] = 50; // change the value of the element BEFORE the middle element to 50
nums[midN] = 100; // change the value of the MIDDLE element to 100
nums[midN+1] = 150; // change the value of the element AFTER the middle element to 150
```

Position	1st	2nd	3rd	4th	5th	6th	7th
nums	8	18	50	100	150	35	89
	nums[0]	nums[1]	nums[midN-1]	nums[midN]	nums[midN+1]	nums[5]	nums[6]

```
String[] txt = { "We", "the", "people", "of", "the", "United", "States" };
int lenT = txt.length; // 7
int midT = lenT / 2; // 7 ÷ 2 = 3 (integer division! Throw away the fraction!)
int middle = txt[midT]; // "of"  (TRUE middle element)
int beforeMiddle = txt[midT-1]; // "people"  (the element BEFORE the middle element)
int afterMiddle = txt[midT+1]; // "the"  (the element AFTER the middle element)
```

NOTE: The element at position `mid` is the **true middle element of the array**.
Notice an equal number of elements on either side of this middle element.

Position	1st	2nd	3rd	4th	5th	6th	7th
txt	"We"	"the"	"people"	"of"	"the"	"United"	"States"
	txt[0]	txt[1]	txt[midT-1]	txt[midT]	txt[midT+1]	txt[5]	txt[6]
Index (position)	0	1	2 mid-1	3 mid	4 mid+1	5	6

```
txt[midT-1] = "fish"; // change the value of the element BEFORE the middle element to s"fish";
txt[midT] = "and"; // change the value of the MIDDLE element to "and"
txt[midT+1] = "chips"; // change the value of the element AFTER the middle element to "chips"
```

Position	1st	2nd	3rd	4th	5th	6th	7th
txt	"We"	"the"	"fish"	"and"	"chips"	"United"	"States"
	txt[0]	txt[1]	txt[midT-1]	txt[midT]	txt[midT+1]	txt[5]	txt[6]

ARRAY-1 BASICS

Accessing values of individual array elements

public int **getFirstElement1**(int[] list)

Given an array of integers values, return the value of the first element

An array / list is a collection of elements, all of the same type.
int[] list = {1,2,3} creates a list of ints (integers).
String[] list = {"To","be","or","not","to","be"} creates a list of Strings.

The way you access an individual element of a list is by its **index / position**.
The syntax is:
(a) **the name of the list variable** followed by
(b) the **index number inside of square brackets**.
For example:

nums[0] is the 1st element of the array variable int[] **nums**.
list[1] is the 2nd element of the array variable int[] **list**.
words[2] is the 3rd element of the array variable String[] **words**.

As with Strings, you can use expressions instead of hard-coded numbers.
For example:

int[] values;
int **len** = values.length; // NOTE: no parentheses after **length**
values[len-1] is the last element in **values**
values[len-2] is the 2nd-to-last element in **values**

```
public int getFirstElement(int[] list) {
  int first = list[0];
  return first;
}
```

getFirstElement1([1, 2, 3, 4, 5]) → 1
getFirstElement1([5, 10, 15, 20]) → 5
getFirstElement1([3, 6, 9]) → 3

ARRAY-1 BASICS

public int **getLastElement1**(int[] list)

If the arrays you're testing all have different lengths,
then you have to access the last element
by using an expression for the index/position:
list[**len-1**].

```
public int getLastElement(int[] numberList) {
  int len = numberList.length;
  int last = numberList[len-1];
  return last;
}
```

public int **getLastElement2**(int[] nums)

All arrays will have 4 elements.

If the arrays you're testing all have the **same** length,
you don't have to use the expression **[len-1]**.
You can just use the position number itself.

For example:
If the list has 4 elements, use **nums[3]** for int[] nums.
If the list has 10 elements, use **words[9]** for String[] words.
If the list has 3 elements, use **list[2]** for int[] list.

```
// if the length of the arrays are ALL = 4
public String getLastElement2(String[] words) {
  int len = words.length;
  String lastOf4 = words[3];
  return lastOf4;
}
```

OR

```
// if the arrays have different lengths
public String getLastElement2(String[] words) {
  int len = words.length;
  String last = words[len-1];
  return last;
}
```

ARRAY-1 BASICS

Comparing values of individual array elements

public boolean **sameFirstLast1**(int[] nums)

Given a bunch of arrays, all of whose lengths are 4,
return true if the value of the 1st and last elements are the same.

```
public boolean sameFirstLast1(int[] nums) {
  boolean same = false;
  int first = nums[0];
  int lastOf4 = nums[3];
  if (first == lastOf4) {
    same = true;
  }
  return same;
}
```

public boolean **sameFirstLast2**(int[] nums)

Given a bunch of arrays, all with **DIFFERENT** lengths,
return true if the value of the 1st and last elements are the same.

```
public boolean sameFirstLast2(int[] nums) {
  boolean same = false;
  int len = nums.length;
  int first = nums[0];
  int last = nums[len-1];
  if ( first == last) {
    same = true;
  }
  return same;
}
```

ARRAY-1 BASICS

Adding elements to get a sum

public int **sumFirstTwo**(int[] nums)

Given an array, return the sum of the 1st two elements.

Strategy: Create a variable **sum**.
Use an assignment statement (=).
On the right side, add the **VALUES** of the first two elements.

```
public int sumFirstTwo(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int sumFirst2 = first + second;
  return sumFirst2;
}
```

Comparing sums

public boolean **sumOf1stAnd2ndEquals3rd**(int[] nums)

Given an array, all of whose lengths are 3,
return true if the sum of the 1st 2 elements equals the value of the 3rd.

Create a variable called **sum** and assign it the sum of the first two elements.
Use an if statement to check whether the sum == the 3rd element.

```
public boolean sumOf1stAnd2ndEquals3rd(int[] nums) {
  boolean equals = false;
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  int sumFirst2 = first + second;
  if ( sumFirst2 == third ) {
    equals = true;
  }
  return equals;
}
```

ARRAY-1 BASICS

Multiplying the VALUE of elements

public int **getTwice1stValue**(int[] nums)

Given an array, all of whose lengths are at least 1,
return **twice** the **value** of the first element.

```
public int getTwiceFirstValue(int[] nums) {
  int first = nums[0];
  int twiceFirst = first * 2;
  return twiceFirst;
}
```

public int **getTwiceSumOf1stAnd2nd**(int[] nums)

Given an array, all of whose lengths are at least 2,
return **twice** the **sum** of the first and second elements.

Create a variable called **sum** and assign it the sum of the first two elements.
Multiply **sum** by 2.
Return that product.

```
public int getTwiceSumOf1stAnd2nd(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int sum = first + second;
  int twiceSumFirst2 = sum * 2;
  return twiceSumFirst2;
}
```

ARRAY-1 BASICS

Two input arrays

public boolean **firstElementsEqual**(int[] a, int[] b)

Given two arrays - **a** and **b** - both of whose lengths are at least 1,
return true if their first elements are the same.

```
public boolean firstElementsEqual(int[] a, int[] b) {
  boolean equals = false;
  int firstA = a[0];
  int firstB = b[0];
  if ( firstA == firstB ) {
    equals = true;
  }
  return equals;
}
```

public boolean **sumAEqualsSumB**_2(int[] a, int[] b)

Given two arrays - **a** and **b** - both of whose lengths are 2,
return true if the sum of **a**'s elements equals the sum of **b**'s elements.

```
public boolean sumAEqualsSumB_2(int[] a, int[] b) {
  boolean equals = false;
  int firstA = a[0];
  int secondA = a[1];
  int firstB = b[0];
  int secondB = b[1];
  int sumA = firstA + secondA;
  int sumB = firstB + secondB;
  if ( sumA == sumB ) {
    equals = true;
  }
  return equals;
}
```

ARRAY-1 BASICS

Comparing elements to see whether they are in Ascending Order

public boolean **inOrder2**(int[] nums)

Given an array with 2 elements
return true if they are in ascending order.

```
public boolean inOrder2(int[] nums) {
  boolean inorder = false;
  int first = nums[0];
  int second = nums[1];
  if ( first < second ) {
    inorder = true;
  }
  return inorder;
}
```

public boolean **inOrder3**(int[] nums)

Given an array with 3 elements
return true if they are in ascending order.

```
public boolean inOrder3(int[] nums) {
  boolean inorder = false;
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  if ( first < second   &&   second < third ) {
    inorder = true;
  }
  return inorder;
}
```

ARRAY-1 BASICS

In Consecutive Order

public boolean **inConsecutiveAscendingOrder2**(int[] nums)

Given an array with 2 elements
return true if they are in CONSECUTIVE ascending order.

Consecutive order means that the next element differs from
the previous element by 1.

```
public boolean inConsecutiveAscendingOrder2(int[] nums) {
  boolean inorder = false;
  int first = nums[0];
  int second = nums[1];
  if ( first + 1 == second ) {
    inorder = true;
  }
  return inorder;
}
```

public boolean **inConsecutiveAscendingOrder3**(int[] nums)

Given an array with 3 elements
return true if they are in CONSECUTIVE ascending order.

```
public boolean inConsecutiveAscendingOrder3(int[] nums) {
  boolean inorder = false;
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  if ( first + 1 == second   &&   second + 1 == third ) {
    inorder = true;
  }
  return inorder;
}
```

ARRAY-1 BASICS

public boolean **inConsecutiveDescendingOrder3**(int[] nums)

Given an array with 3 elements
return true if they are in CONSECUTIVE Descending order.

```
public boolean inConsecutiveDescendingOrder3(int[] nums) {
  boolean inorder = false;
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  if ( first - 1 == second   &&   second - 1 == third ) {
    inorder = true;
  }
  return inorder;
}
```

ARRAY-1 BASICS

Finding the LARGEST or SMALLEST element

public int **getSmallest2**(int[] nums)

Given an int[] array with 2 elements
return the **smallest** value.

Create an **int return** variable called **smallest**.
Assign it a **default** value, e.g. the first element.
Use an if statement to check if the other value is smaller -
and if it is smaller, **change** the value of **smallest** to that value.

```
public int getSmallest2(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int smallest = first;
  if ( second < smallest ) {
    smallest = second;
  }
  return smallest;
}
```

Note: Another way to check for the smaller of **two** values is to use **Math.min()**:

```
public int getSmallest2(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int smallest = Math.min ( first, second );
  return smallest;
}
```

ARRAY-1 BASICS

public int **getSmallest3**(int[] nums)

Given an int[] array with 3 elements
return the **smallest** value.

Create an **int return** variable called **smallest**.
Assign it a **default** value, e.g. the first element.

Use an if statement to check if the 2nd value is smaller -
and if it is smaller, **change** the value of **smallest** to that value.

Use ANOTHER if statement to check if the 3rd value is smaller -
If it is, **change** the value of **smallest** to the 3rd element.

```
public int getSmallest3(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  int smallest = first;
  if ( second < smallest ) {
    smallest = second;
  }
  if (third < smallest ) {
    smallest = third;
  }
  return smallest;
}
```

Below is how you would do the same thing using Math.min().

```
public int getSmallest3(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  int smallest12 = Math.min( first, second );
  int smallest123 = Math.min( smallest12, third );
  return smallest123;
}
```

ARRAY-1 BASICS

public int **getSmallest4**(int[] nums)

Given an int[] array with 4 elements
return the **smallest** value.

Create an **int return** variable called **smallest**.
Assign it a **default** value, e.g. the first element.

Use an if statement to check if the 2nd value is smaller -
and if it is smaller, **change** the value of **smallest** to that value.

Use ANOTHER if statement to check if the 3rd value is smaller -
If it is, **change** the value of **smallest** to the 3rd element.

Use a 3rd if statement to check if the 4th value is smaller -
If it is, **change** the value of **smallest** to the 4th element.

```
public int getSmallest4(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  int fourth = nums[3];
  int smallest = first;
  if (second < smallest) {
    smallest = second;
  }
  if (third < smallest) {
    smallest = third;
  }
  if (fourth < smallest) {
    smallest = fourth;
  }
  return smallest;
}
```

Below is how you would do the same thing using Math.min().

```
public int getSmallest4(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int third = nums[2];
  int fourth = nums[3];
  int smallest12 = Math.min( first, second );
  int smallest123 = Math.min( smallest12, third );
  int smallest1234 = Math.min( smallest123, fourth );
  return smallest1234;}
```

ARRAY-1 BASICS

public int **getLargest2**(int[] nums)

Given an int[] array with 2 elements
return the **largest** value.

Create an **int return** variable called **largest**.
Assign it a **default** value, e.g. the first element.
Use an if statement to check if the other value is larger -
and if it is larger, **change** the value of **largest** to that value.

```
public int getLargest2(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int largest = first;
  if (second > largest) {
    largest = second;
  }
  return largest;
}
```

Note: Another way to check for the larger of **two** values is to use **Math.max()**:

```
public int getLargest2(int[] nums) {
  int first = nums[0];
  int second = nums[1];
  int largest = Math.max( first, second );
  return largest;
}
```

ARRAY-1 BASICS

public int **getSmallest2_DifferentLengthsA**(int[] nums)

Given an int[] array whose length is anywhere from 0-2,
i.e. it has either zero, one or two elements,
return the **smallest** value.

This problem requires that you check the length of the array
before you try to access an element, to make sure that it exists.

Therefore, you will have 3 different if statements - 3 different sections -
one for each of the possible array lengths.

```
public int getSmallest2_DifferentLengths(int[] nums) {
  int smallest = 0; // default if length == 0 (the empty array)
  int len = nums.length;

  if (len == 0) {
    smallest = 0; // NOT NECESSARY, because length == 0 is the default
  }

  if (len == 1) {
    smallest = nums[0];
  }

  if (len == 2) {
    smallest = Math.min( nums[0], nums[1] );
  }

  return smallest;
}
```

Note that the three sections are **mutually exclusive**.
That means:

An **empty array** will only execute the statement(s) within the
block governed by **if (len == 0)**

An array with **a single element** will only execute the statement(s)
within the block governed by **if (len == 1)**

An array with **2 elements** will only execute the statement(s)
within the block governed by **if (len == 2)**

ARRAY-1 BASICS

public int **getSmallest3_DifferentLengthsA**(int[] nums)

Given an int[] array whose length is anywhere from 0-3,
i.e. it has either zero, one, two or three elements,
return the **smallest** value.

This problem requires that you check the length of the array
before you try to access an element, to make sure that it exists.

Therefore, you will have 4 different if statements - 4 different sections -
one for each of the possible array lengths.

```
public int getSmallest3_DifferentLengthsA(int[] nums) {
  int smallest = 0; // default if length == 0 (the empty array)
  int len = nums.length;

  if (len == 0) {
    smallest = 0; // NOT NECESSARY, because length == 0 is the default
  }

  if (len == 1) {
    smallest = nums[0];
  }

  if (len == 2) {
    smallest = Math.min( nums[0], nums[1] );
  }

  if (len == 3) {
    smallest = Math.min( nums[0], nums[1] );
    smallest = Math.min( smallest, nums[2] );
  }

  return smallest;
}
```

public int **getSmallest3_DifferentLengthsB**(int[] nums)

Given an int[] array whose length is anywhere from 0-3, i.e. it has either zero, one, two or three elements, return the **smallest** value.

This is the same kind problem as getSmallest2_DifferentLengths(),
but solved a bit differently.

As in that problem, there are different sections for arrays of different lengths.
The difference, however, is that they are **NOT mutually exclusive**.
More than one if statement will be executed, depending upon the length of the array.

```
public int getSmallest3_DifferentLengthsB(int[] nums) {
  int smallest = 0; // default if len == 0
  int len = nums.length;

  if (len >= 0) {
    smallest = 0;
  }

  if (len >= 1) {
    smallest = nums[0];
  }

  if (len >= 2) {
    smallest = Math.min( smallest, nums[1] );
  }

  if (len >= 3) {
    smallest = Math.min( smallest, nums[2] );
  }

  return smallest;
}
```

Note how what we call **the flow of control** works in this solution.

Arrays of all lengths will execute the statement(s) within the
block governed by **if (len >= 0)**

Arrays with lengths > 0 will execute the statement(s)
within the block governed by **if (len >= 1)**

Arrays with lengths > 1 will execute the statement(s)
within the block governed by **if (len >= 2)**

ARRAY-1 BASICS

Arrays with lengths > 2 will execute the statement(s)
within the block governed by **if (len >= 3)**

Notice that the longer the array, the more sections it will execute,
in a cascading sort of fashion.