

# Arrays

# What is an Array?

An array is a simple **data structure** that is used to store a set of variables of the same data types

# Why is it necessary?

- Suppose we need to write a Java program that processes a set of variables (e.g., grades of 10 students):

- Using Variables:

- Declare 50 integer variables each represents a certain student's grade
  - Example: `int grade1, grade2, grade3.....grade50;`
- How many variables will you need for the whole University?!

- Using Arrays:

- Declare an array that has the same size as the number of individual variables
  - primitive data types
  - object data types
- You can access a specific grade in the array using corresponding student's index:

# Arrays

- Arrays make writing programs much easier than using a large number of variables
- An Array is a fixed-size random-access data structure:
  - **fixed-size**: you have to specify size of array at creation time
  - **random-access**: allows inserting and reading any item in any order

# Examples of Arrays in Real Life

1	2	3	4
---	---	---	---



	0	1	2
0	.392	.482	.576
1	.478	.63	.169
2	.580	.79	.263
	0	1	2
0	.376	.443	.569
1	.306	.478	.561
2	.263	.44	.60

# Imagine you need to build a tray....

- What type of tray is it?
  - What will it hold? Eggs? Cakes? Etc
  - What will be its maximum capacity?

# Array Definition & Declaration

- Array = data structure that represents a collection of the **same type of data**.
- Example:
  - Grades for students in a class
  - US state names

# Array Definition & Declaration

Array = data structure that represents a collection of the same type of data.

<u>          </u> data-type	[ ]	<u>                    </u> variable name	= new	<u>                    </u> data-type	[ <u>      </u> ]	;
					size	



# Example

- Suppose we need to write a Java program that processes a set of variables (e.g., grades of 10 students):
  - Using Variables:
    - Example: `int grade1, grade2, grade3.....grade10;`
  - Using Arrays:
    - primitive data types: `int grades[] = new int[10];`
    - object data types: `Student studentList[] = new Student[10];`
    - You can access a specific grade in the array using corresponding student's index:
      - `grades[5] = 99`
      - `studentList[5] = new Student("John", 10);`

# Primitive variables vs. Reference Variables

```
int grades[10];
```

grades
10
20
5
0
6
18
13
20
14
5

```
Student list[10];
```

studentList	
	John, 10
	Eric, 20
	Smith, 5
	Lina, 0
	Der, 6
	Xyz, 18
	Sam, 13
	Pat, 20
	Dave, 14
	Jim, 5

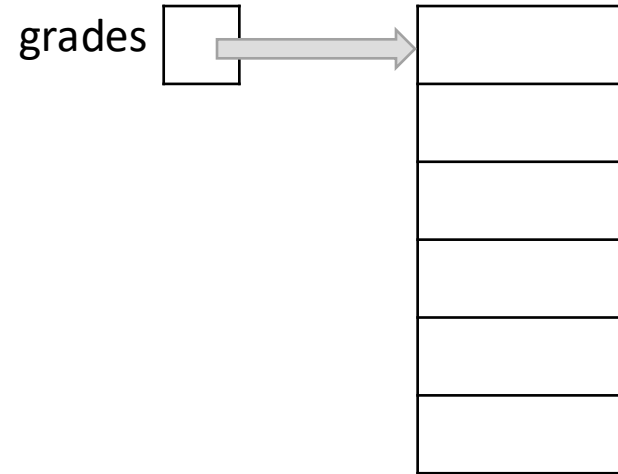
# Declaring an Array Reference Variable

```
int grades[];
```

grades 

# Instantiating an Array

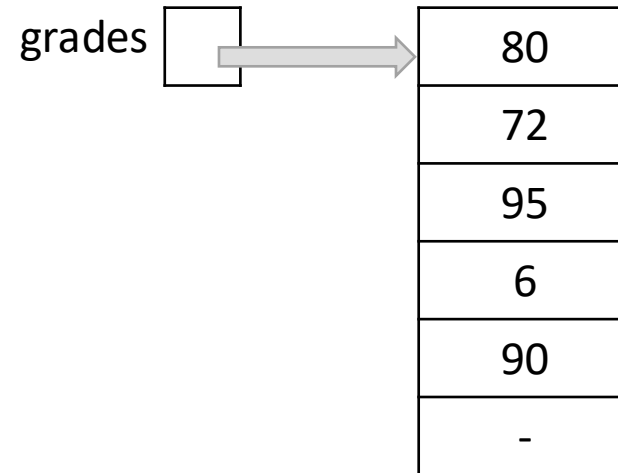
```
int grades[] = new int[6];
```



# Initializing the Contents of an Array

```
grades[0] = 80;
```

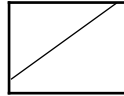
```
grades[1] = 72;
```



# Declaring an Array of Objects

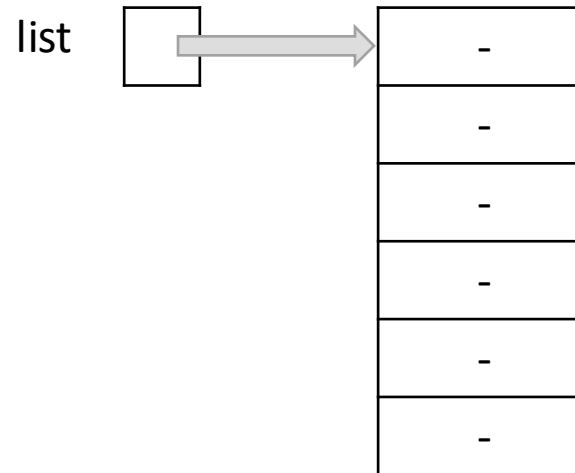
```
Student[] list;
```

list



# Instantiating an Array of Objects

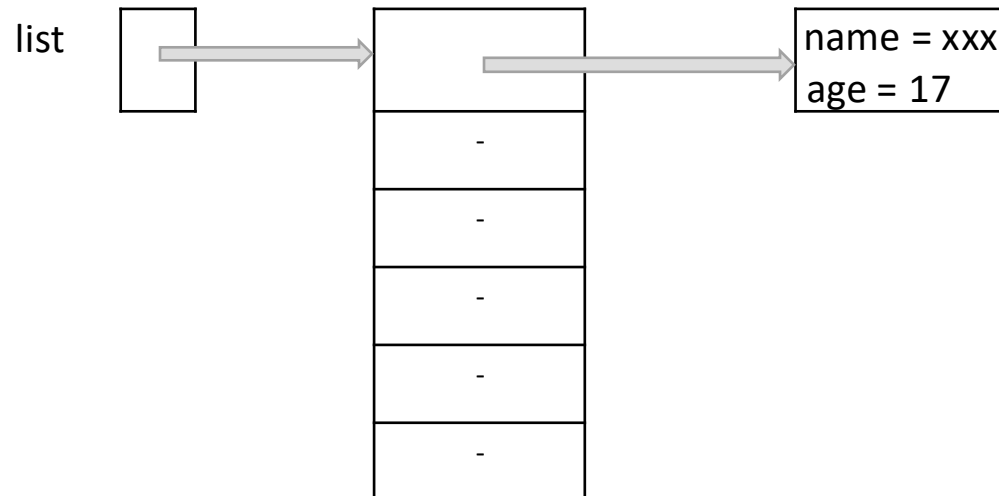
```
Student[] list = new Student[6];
```



- Only the array has been instantiated. No `Student` objects have been created yet.

# Initializing an Array of Objects

```
Student s1 = new Student("abc",30);  
Student s2 = new Student("xyz", 20);  
list[0] = new Student("xxx", 17);
```





# Properties of simple Arrays

- **Data is not sorted**
- **Search operation is costly:**
  - You have to go through all elements in the array until you find your required item
  - Can we implement Sorted Arrays?
    - Search is more efficient
    - Insertion operation is costly (why?)
- **Deletion is costly** in both sorted and unsorted arrays
  - You need to move on average half the items to fill in the gap.
- **Fixed size:**
  - You may not know the number of data items at the beginning of the program
  - What if you estimate array size?
    - If too large → you are wasting memory
    - If too small → your program will fail

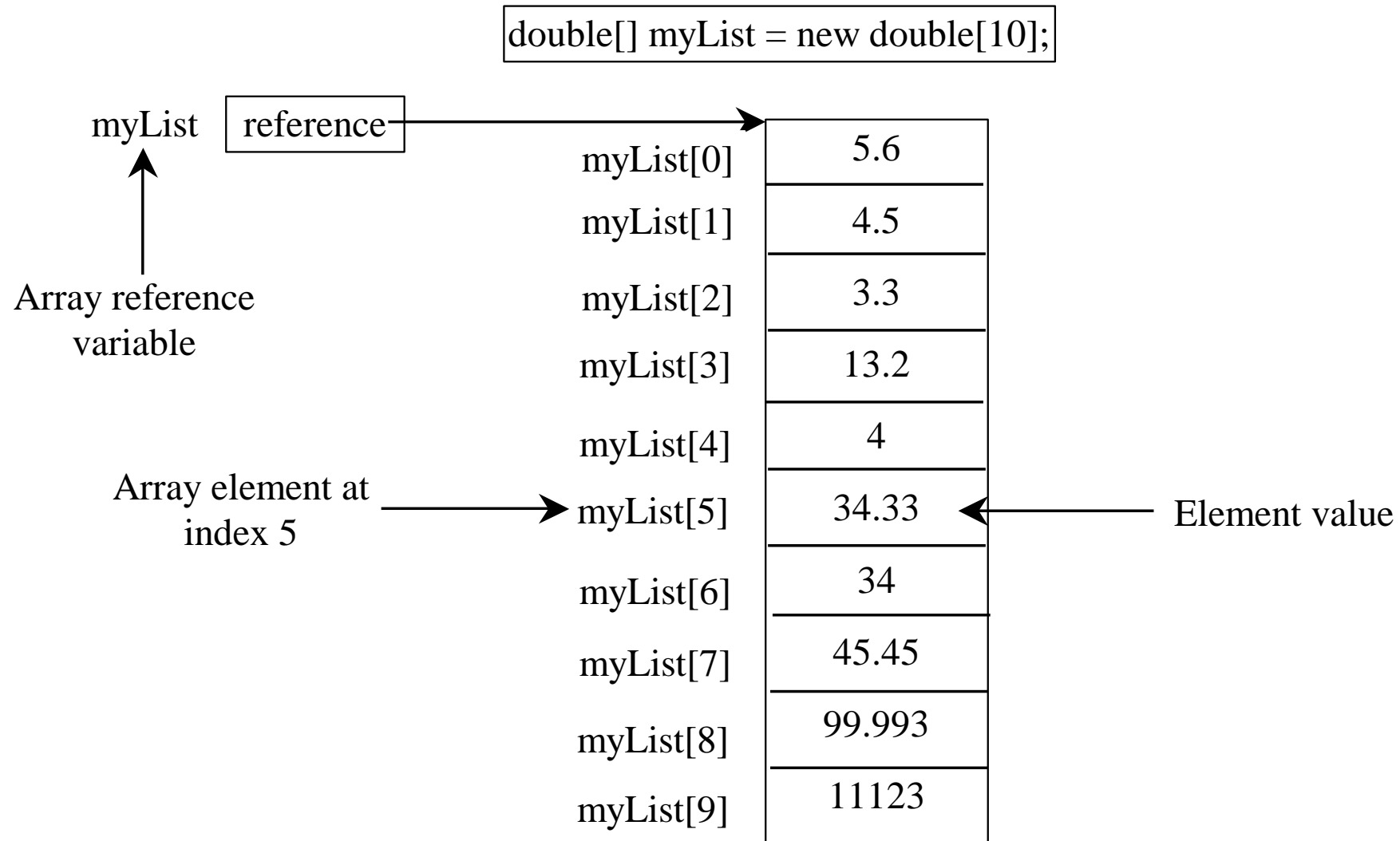
# Capacity != Number of Items in Array

- The capacity of the array is determined when the array is declared. This array has capacity 6.

```
Student[] list = new Student[6];
```

- The number of items stored in the array may be less than the capacity of the array
- In the previous slide, there is only 1 item stored in an array with the capacity to hold 6
- `list.length` refers to the **capacity** of the array and NOT the number of elements stored in the array

# Example



# Thoughts on arrays

- Can the length of the array ever change?

No. Once created, an array size is fixed and cannot be changed

- Can the number of things stored in the array change?

- Yes. We can keep track of the number of items using a *numItems* variable

- How do we access items in the array?

We use array **indexing**: each slot in the array has an index, starting with **index 0**

# Manipulating Elements of the Array

- Array elements are used in the same way as variables

```
int[] myArr = new int[20];  
myArr[0] = 10*5 + 3;  
System.out.println(myArr[0] );
```

- The element's position can be calculated using any mathematical operations:

```
int x = 3, y = 5;  
myArr[y] = 10;  
myArr[y-2] = 20;  
myArr[y-x] = myArr[1] - myArr[2];
```

# Array Indexing and Array Bounds Rules

- When writing a loop to go over all elements in the array, make sure:
  - Array subscript **never goes below 0**
  - Maximum array subscript should be **< length**

<code>int[] data = new int[10];</code>	
<code>data[ -1 ]</code>	<b>always illegal</b>
<code>data[ 10 ]</code>	<b>illegal</b> (given the above declaration)
<code>data[ 1.5 ]</code>	<b>always illegal</b>
<code>data[ 0 ]</code>	<b>always OK if the array exists</b>
<code>data[ 9 ]</code>	<b>OK</b> (given the above declaration)
<code>data[ j ]</code>	<b>can't tell without more information</b> (depends on the current value of j)

# Common Things we do with arrays

- Fill the array
- Modify (add/remove)
- Print
- Search for something
- Sort values in the array
- Accumulate values/compare values

# Common Things we do with arrays

- Fill the array -- Initialization
- Modify (add/remove) -- Insertion/Deletion
- Print -- Traversal & Printing
- Search for something -- Searching
- Sort values in the array -- Sorting
- Accumulate values/compare values



# Filling an array

- Using Initialization list
- Using user input

# Filling an array

- Create an empty array
  - Null
  - zero
- Create with values
- Put values in with a loop
- Put values in one at a time (add an element)

\*Sentinel values or actual values

# When create an array - filled with default value

- Null value – means nothing is there. Can't tell a non-existent thing to do something.
- Default values:
  - 0 for numeric primitive data types
  - '\u0000' for char
  - false for boolean
- NOTE: Empty string is not null.

# Array Initialization using a list of values

```
int[] data = {10, 12, 33, 14, 25};
```

The Java compiler creates an array where the size equals to the number of entries in the initializer's list.

**data**

10
12
33
14
25

# Array Initialization with User Input

//making an object named "input" from the Scanner class. System.in ( the computer input stream)  
is the parameter to the constructor of the Scanner class

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

// myList.length = number of items to put in the array of double myList

```
System.out.print("Enter " + myList.length + " values: ");
```

// the nextDouble() picks each number provided by the user and enters it in the array at position i

```
for (int i = 0; i < myList.length; i++)  
    myList[i] = input.nextDouble();
```

# Initializing arrays with input values

```
java.util.Scanner input = new java.util.Scanner(System.in);  
  
System.out.print("Enter " + myList.length + " values: ");  
  
for (int i = 0; i < myList.length; i++)  
    myList[i] = input.nextDouble();
```

Deleting from an Array

# How to delete an item from the array?

- You cannot delete an item from the array, however, you can replace the value that you do not want any more with another value.
- Two ways to do that:
  - start from the index after the value to delete, shift all elements down to overwrite the item to be deleted.
  - Replace the element to be deleted with the last element in the array.



# Array Traversal

Example: what is the output of the following piece of code?

```
int[] values = new int[5];

for (int i = 1; i < 5; i++) {
    values[i] = i + values[i-1];
}

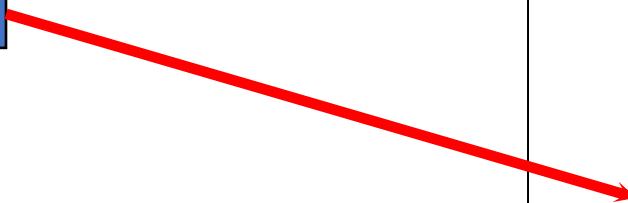
values[0] = values[1] + values[4];

for (int i = 0; i < 5; i++) {
    System.out.println(i+"\\t"+values[i]);
}
```

Declare an array variable. An array is created in memory and all values are initialized to 0


```
for (int i = 1; i < 5; i=i+1) {  
    values[i] = i + values[i-1];  
}  
  
values[0] = values[1] + values[4];  
  
for (int i = 0; i < 5; i=i+1) {  
    System.out.println(i+"\\t"+values[i]);  
}
```

values



0	0
1	0
2	0
3	0
4	0

**i = 1**


```
int[] values = new int[5];  
  
for (int i  5; i=i+1) {  
    values[i] = i + values[i-1];  
}  
  
values[0] = values[1] + values[4];  
  
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\\t"+values[i]);  
}
```

values

0	0
1	0
2	0
3	0
4	0

i (=1) is less than 5

```
int[] values = new int[5];
```

```
for (int i = 1; i < ) {  
    values[i] = i + values[i-1];  
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\t"+values[i]);  
}
```

values

0	0
1	0
2	0
3	0
4	0

After this line is executed, `values[1] = 1`

```
int[] values = new int[5];
```

```
for (int i = 1; i < 5; i=i+1) {
```

```
    valu
```

```
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\t"+values[i]);
```

```
}
```

values

0	0
1	
2	0
3	0
4	0

After  $i=i+1$ ,  $i = 2$

```
int[] values = new int[5];
```

```
for (int i = 1; i < 5; i=i+1)
```

```
values[i] = i + values[i-1];
```

```
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\t"+values[i]);
```


```
}
```

values

0	0
1	1
2	0
3	0
4	0

i (= 2) is less than 5

```
int[] values = new int[5];
```

```
for (int i = 1; i < ) {  
    values[i] = i + values[i-1];  
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\\t"+values[i]);  
}
```

values

0	0
1	1
2	0
3	0
4	0



After this line is executed,  
`values[2] = 3 (2 + 1)`

```
int[] values = new int[5];
```

```
for (int i = 1; i < 5; i=i+1) {
```

```
value
```

```
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\t"+values[i]);  
}
```

values

0	0
1	1
2	
3	0
4	0

After this line, `values[3] = 6 (3 + 3)`

```
int[] values = new int[5];
```

```
for (int i = 1; i < 5; i=i+1) {
```

```
    values[i] = values[i-1] + values[i-2];
```

```
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\\t"+values[i]);
```

```
}
```

values

0	0
1	1
2	3
3	6
4	0

After this, `values[4] = 10 (4 + 6)`

```
int[] values = new int[5];
```

```
for (int i = 1; i < 5; i=i+1) {
```

```
    val
```

```
}
```

```
values[0] = values[1] + values[4];
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\t"+values[i]);
```

```
}
```

values

0	0
1	1
2	3
3	6
4	

After this line, `values[0] = 11 (1 + 10)`

```
int[] values = new
```

```
for (int i = 1; i < 5; i=i+1) {
```

```
values[i] = 1 + values[i-1];
```

```
}
```

```
for (int i = 0; i < 5; i=i+1) {    System.out.println(i+"\\t"+values[i]);
```

```
}
```

values

0	
1	1
2	3
3	6
4	10

# Example: Summing all elements

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
    total += myList[i];
}
```

# Example: Finding the largest element

```
double max = myList[0];  
  
for (int i = 1; i < myList.length; i++) {  
    if (myList[i] > max)  
        max = myList[i];  
}
```

# Activity

- Write the code to find the position of the smallest element in an array of integers

# Shifting Elements in an array

```
double temp = myList[0]; // Retain the first element
```

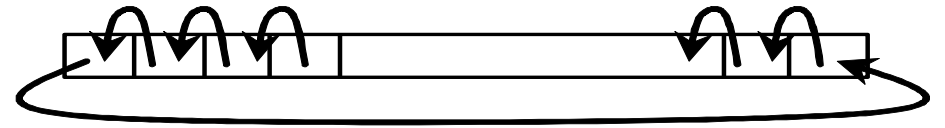
```
// Shift elements left
```

```
for (int i = 1; i < myList.length; i++) {  
    myList[i - 1] = myList[i];  
}
```

```
// Move the first element to fill in the last position
```

```
myList[myList.length - 1] = temp;
```

myList





# Printing arrays

```
for (int i = 0; i < myList.length; i++) {  
    System.out.print(myList[i] + " ");  
}
```

# The Enhanced for Loop (for-each)

- Simplified array processing (read only). Always goes through all elements
- General format:

```
for(datatype elementVariable : arrayName)  
    statement;
```

- Example:

```
int[] numbers = {3, 6, 9};  
For(int val : numbers){  
    System.out.println("The next value is " + val); }
```

- Another Example:

```
for (double value: myList)  
    System.out.println(value);
```

**Note:** You still have to use an index variable if you wish to traverse the array in a different order or change the elements in the array.

# Multi-Dimensional Arrays

# Motivations

- Thus far, you have used one-dimensional arrays to model linear collections of elements.
- You can use a two-dimensional array to represent a matrix or a table.
- For example, the following table that describes the distances between the cities can be represented using a two-dimensional array.

	Distance Table (in miles)						
	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

# Declare/Create Two-dimensional Arrays

```
// Declare array refVar
```

```
dataType[][] refVar;
```

```
// Create array and assign its reference to variable
```

```
refVar = new dataType[10][10];
```

```
// Combine declaration and creation in one statement
```

```
dataType[][] refVar = new dataType[10][10];
```

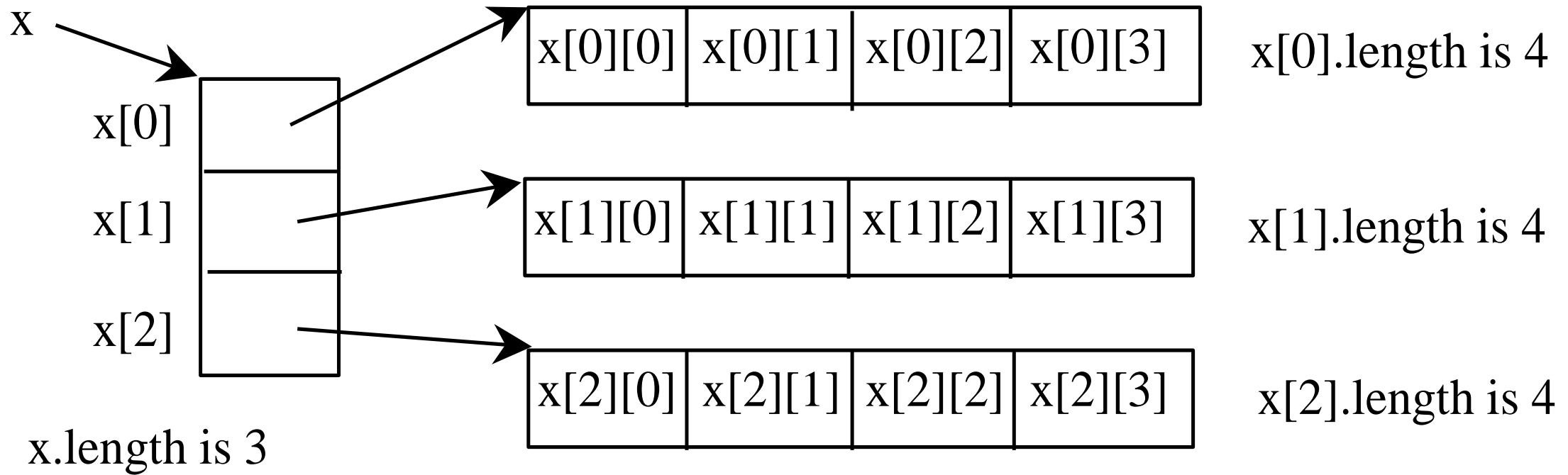
# Declaring Variables of Two-dimensional Arrays and Creating Two-dimensional Arrays

```
int[][] matrix = new int[10][10];
```

```
for (int i = 0; i < matrix.length; i++)  
    for (int j = 0; j < matrix[i].length; j++)  
        matrix[i][j] = (int) (Math.random() * 1000);
```

# Lengths of Two-dimensional Arrays

```
int[][] x = new int[3][4];
```



# Two-dimensional Array Illustration

	0	1	2	3	4
0					
1					
2					
3					
4					

```
matrix = new int[5][5];
```

	0	1	2	3	4
0					
1					
2		7			
3					
4					

```
matrix[2][1] = 7;
```

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9
3	10	11	12

```
int[][] array = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {10, 11, 12}  
};
```

- `matrix.length?` 5
- `matrix[0].length?` 5

- `array.length?` 4
- `array[0].length?` 3



## Lengths of Two-dimensional Arrays, cont.

```
int[][] array = {  
    {1, 2, 3},  
    {4, 5, 6},  
    {7, 8, 9},  
    {10, 11, 12}  
};
```

```
array.length  
array[0].length  
array[1].length  
array[2].length  
array[3].length
```

```
array[4].length
```

**ArrayIndexOutOfBoundsException**

# Initializing arrays with input values

```
java.util.Scanner input = new Scanner(System.in);
```

```
System.out.println("Enter " + matrix.length + " rows and " + matrix[0].length + " columns:");
```

```
for (int row = 0; row < matrix.length; row++) {  
    for (int column = 0; column < matrix[row].length; column++) {  
        matrix[row][column] = input.nextInt();  
    }  
}
```

# Initializing arrays with random values

```
for (int row = 0; row < matrix.length; row++) {  
    for (int column = 0; column < matrix[row].length; column++) {  
        matrix[row][column] = (int)(Math.random() * 100);  
    }  
}
```

# Ragged Arrays

Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as *a ragged array*.

For example,

```
int[][] matrix = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```

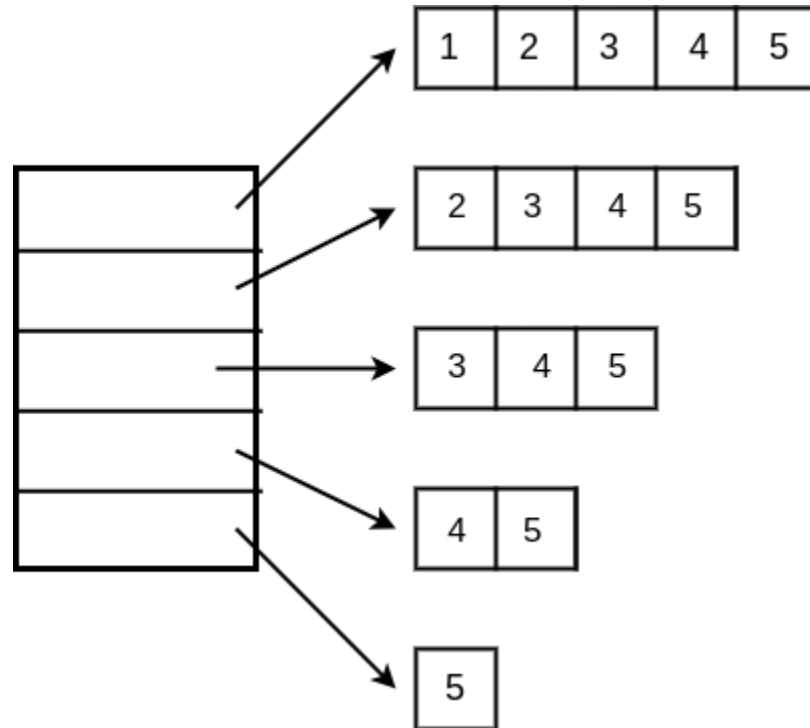
```
matrix.length is 5  
matrix[0].length is 5  
matrix[1].length is 4  
matrix[2].length is 3  
matrix[3].length is 2  
matrix[4].length is 1
```

# Ragged Arrays

Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as *a ragged array*.

For example,

```
int[][] matrix = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```



# Printing arrays

```
for (int row = 0; row < matrix.length; row++) {  
    for (int column = 0; column < matrix[row].length; column++) {  
        System.out.print(matrix[row][column] + " ");  
    }  
  
    System.out.println();  
}
```

# Summing all elements

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
    for (int column = 0; column < matrix[row].length; column++) {
        total += matrix[row][column];
    }
}
```

# Summing elements by column

```
// fix column number
```

```
for (int column = 0; column < matrix[0].length; column++) {  
    int total = 0;
```

```
// then go through each row of array for the same column number
```

```
    for (int row = 0; row < matrix.length; row++)  
    {  
        total += matrix[row][column];  
    }  
    System.out.println("Sum for column " + column + " is " + total);  
}
```



# Summing elements by column

// fix column number

```
for (int column = 0; column < matrix[0].length; column++) {  
    int total = 0;
```

// then go through each row of array for the same column number

```
    for (int row = 0; row < matrix.length; row++)  
    {  
        total += matrix[row][column];  
    }  
    System.out.println("Sum for column " + column + " is " + total);  
}
```

Q: Can you  
sum by row?

# Multidimensional Arrays (dimension $> 2$ )

Occasionally, you will need to represent n-dimensional data structures.

In Java, you can create n-dimensional arrays for any integer n.

The way to declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare n-dimensional array variables and create n-dimensional arrays for  $n \geq 3$ .

For example, the following syntax declares a three-dimensional array variable scores, creates an array, and assigns its reference to scores.

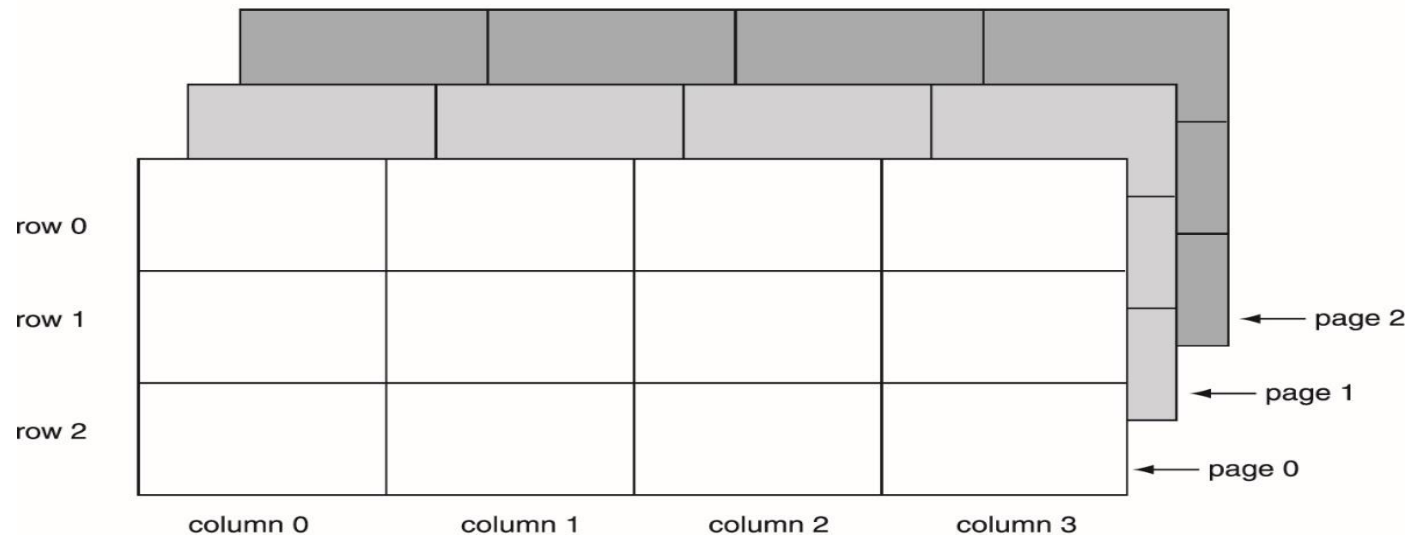
```
double[][][] scores = new double[10][5][2];
```

# Problem: Calculating Total Scores

- Objective: write a program that calculates the total score for students in a class.
- Suppose the scores are stored in a three-dimensional array named scores.
- The first index in scores refers to a student, the second refers to an exam, and the third refers to the part of the exam. (*scores[numStudents][numExams][examPortion]*)
- Suppose there are 7 students, 5 exams, and each exam has two parts--the multiple-choice part and the programming part.
- So, scores[i][j][0] represents the score on the multiple-choice part for the *i*'s student on the *j*'s exam.
- Your program displays the total score for each student.

# More Than Two Dimensions

- Java does not limit the number of dimensions that an array may be.
- More than three dimensions is hard to visualize, but can be useful in some programming problems.



Array of Objects

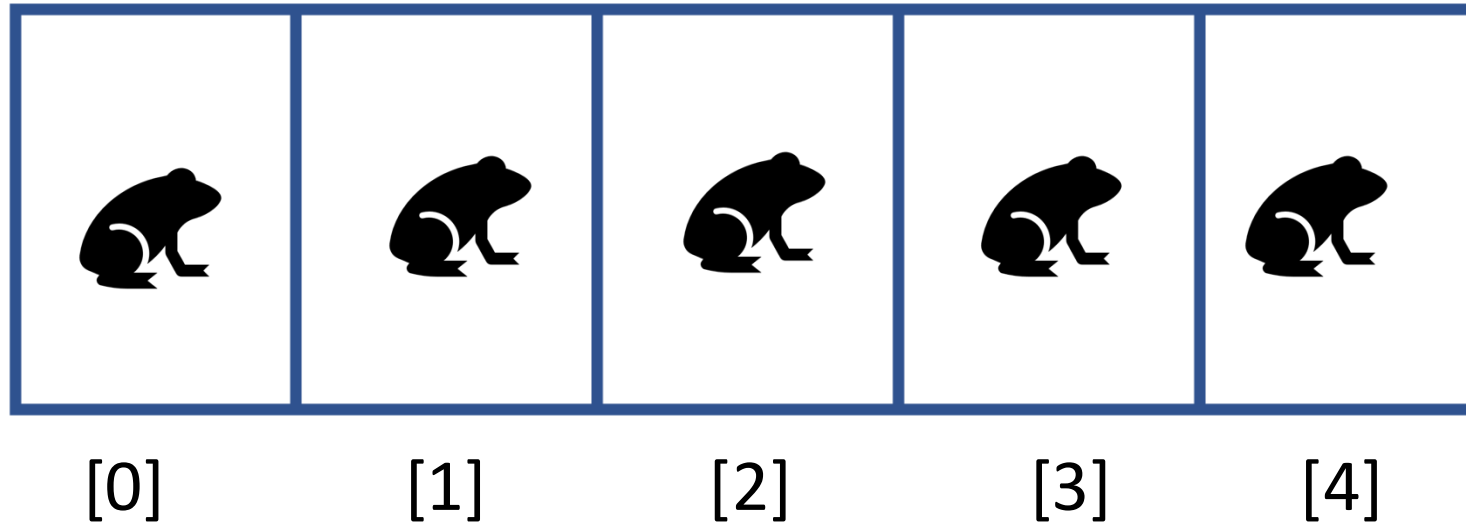
# Accessing Objects in an Array

- We use Array indexing: each location has an index
- Starts at **zero**. (same as for primitives)
- In this case the array is a collection of objects



- Access it by saying “the thing behind door number **three**”.
- Notice first thing is behind door number zero.
- If this array is stored in variable **list**, then the frog is at **list[3]**

# Adding Objects to array



<u>NumItems</u>	<u>Location</u>
0	-
1	0
2	1
3	2
4	3
5	4

Length of array = 5

Array class has a *length* variable and it is public

# Exercise

- Create a `dogArray` static variable in an `AnimalShelter` class
- Create the following two methods in the `AnimalShelter` class.
  - `addAnimal()`: this method accepts an `Dog` object and puts it into the `dog` array. Don't forget to increment the number of animals.
  - `addAnimal()`: an overloaded method that accepts the inputs needed to create a `Dog` instance. Instantiate a `dog` and add it to the array.



# Exercise

- Modify the toString in the Dog class so that it prints the name and age separated by a space.
- Modify the toString for the AnimalShelter class so that it prints all the animals in the shelter (don't delete what is already there, add to it).  
\*\*You are going to be calling the toString of the Dog class to help with this.
- Create a driver class. Inside the main method, create a shelter, add at least one animal, and print the shelter. Experiment with adding more dogs to your shelter and printing the shelter.

# Reassigning & Copying Arrays

# Reassigning Array References (1 of 3)

- An array reference can be assigned to another array of the same type.

```
// Create an array referenced by the numbers variable.
```

```
int[] numbers = new int[10];
```

```
// Reassign numbers to a new array.
```

```
numbers = new int[5];
```

# Reassigning Array References (1 of 3)

- An array reference can be assigned to another array of the same type.

```
// Create an array referenced by the numbers variable.
```

```
int[] numbers = new int[10];
```

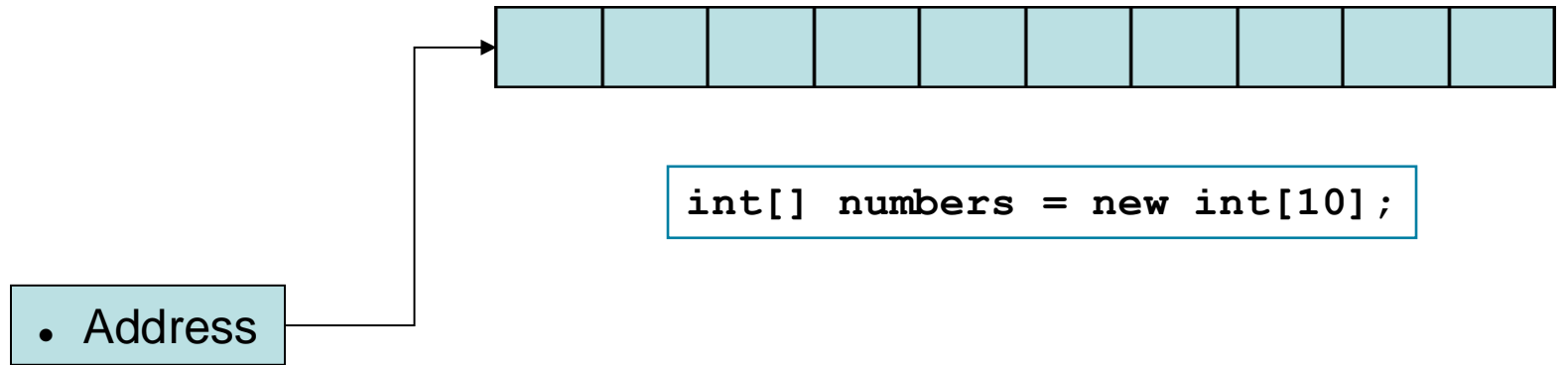
```
// Reassign numbers to a new array.
```

```
numbers = new int[5];
```

← If this first 10-element array no longer has a reference to it, it will be garbage collected.

# Reassigning Array References (2 of 3)

The `numbers` variable holds the address of an `int` array.

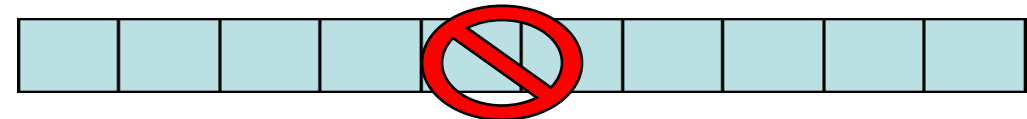


# Reassigning Array References (3 of 3)

- After the assignment statement, *numbers* points to a new reference in the memory.
- The object previously referenced by *numbers* is no longer referenced.
- This object is known as **garbage**.
- Garbage is automatically collected by JVM.

- The `numbers` variable
- holds the address of an `int` array.

• Address



This array gets marked for  
garbage collection

```
numbers = new int[5];
```



# Copying Arrays (1 of 2)

- This is *not* the way to copy an array.

```
int[] array1 = { 2, 4, 6, 8, 10 };
```

```
int[] array2 = array1; // This does not copy array1.
```

array1 holds an

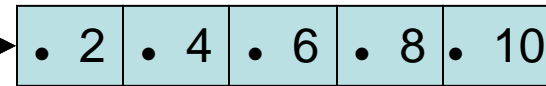
- address to the array

• Address

array2 holds an

- address to the array

• Address



## Copying Arrays (2 of 2)

- You cannot copy an array by merely assigning one reference variable to another.
- You need to copy the individual elements of one array to another.

```
int[] firstArray = {5, 10, 15, 20, 25 };  
int[] secondArray = new int[5];  
for (int i = 0; i < firstArray.length; i++)  
    secondArray[i] = firstArray[i];
```

- This code copies each element of `firstArray` to the corresponding element of `secondArray`.



# Comparing Arrays

- The == operator determines only whether array references point to the same array object.

```
int[] firstArray = { 5, 10, 15, 20, 25 };  
int[] secondArray = { 5, 10, 15, 20, 25 };  
  
if (firstArray == secondArray) // This is a mistake.  
    System.out.println("The arrays are the same.");  
else  
    System.out.println("The arrays are not the same.");
```

# Comparing Arrays (2 of 2) -- Example

```
int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true;
int i = 0;

// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
    arraysEqual = false;

// Next determine whether the elements contain the same data.
while ((arraysEqual) && (i < firstArray.length))
{
    if (firstArray[i] != secondArray[i])
        arraysEqual = false;
    i++;
}

// Print the final value of arrayEqual
if (arraysEqual)
    System.out.println("The arrays are equal.");
else
    System.out.println("The arrays are not equal.");
```

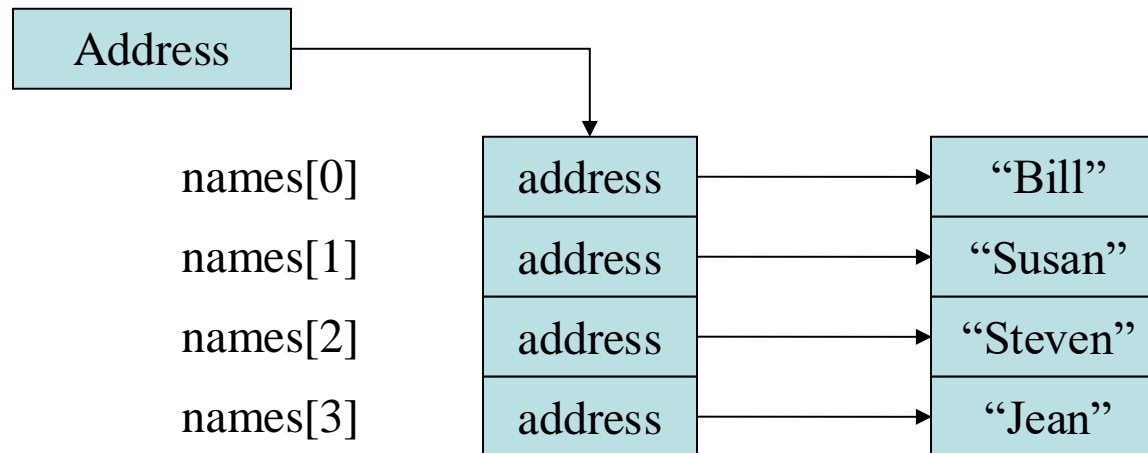
# String Arrays (1 of 3)

- Arrays are not limited to primitive data.
- An array of `String` objects can be created:

```
String[] names = { "Bill", "Susan", "Steven", "Jean" };
```

The `names` variable holds  
the address to the array.

A `String` array is an array  
of references to `String` objects.

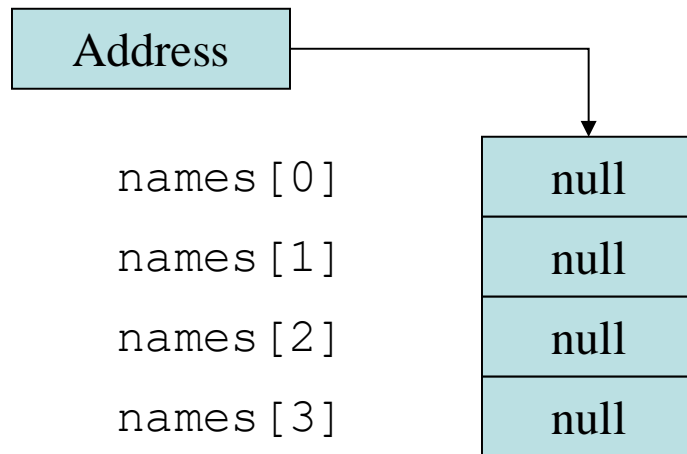


## String Arrays (2 of 3)

- If an initialization list is not provided, the *new* keyword must be used to create the array:

```
String[] names = new String[4];
```

The `names` variable holds  
the address to the array.

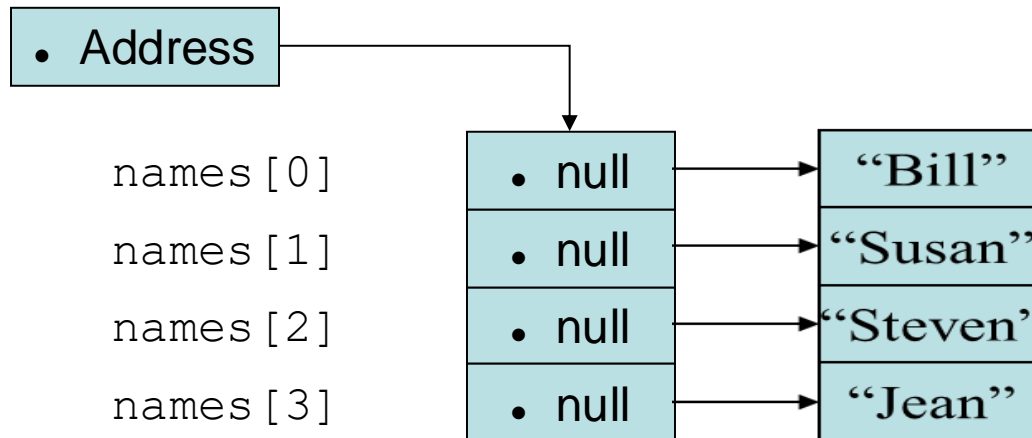


# String Arrays (3 of 3)

- When an array is created in this manner, each element of the array must be initialized.

```
names[0] = "Bill";  
names[1] = "Susan";  
names[2] = "Steven";  
names[3] = "Jean";
```

- The `names` variable holds
- the address to the array.



# Calling String Methods on Array Elements

- `String` objects have several methods, including:
  - `toUpperCase`
  - `compareTo`
  - `equals`
  - `charAt`
- Each element of a `String` array is a `String` object.
- Methods can be used by using the array name and index as before.

```
System.out.println(names[0].toUpperCase());  
char letter = names[3].charAt(0);
```

# The *length* Field and the *length* Method

- Arrays have a **final field** named `length`.
- String objects have a **method** named `length`.
- To display the length of each string held in a `String` array:

```
for (int i = 0; i < names.length; i++)  
    System.out.println(names[i].length());
```

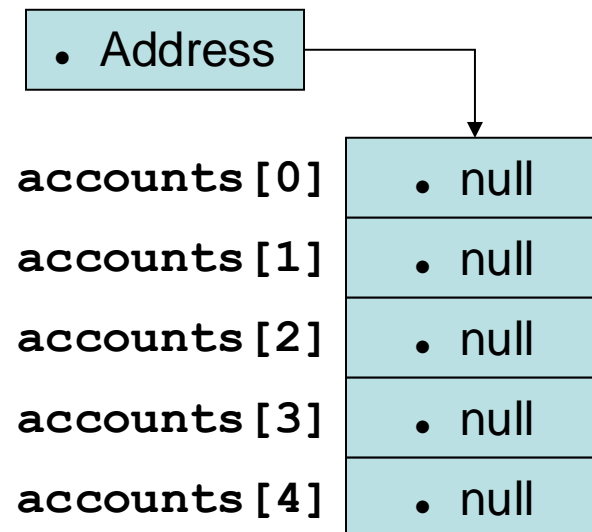
- An array's `length` is a **field**
  - You do not write a set of parentheses after its name.
- A `String`'s `length` is a **method**
  - You do write the parentheses after the name of the `String` class's `length` method.

# Array of Objects (1 of 2)

- An array of Objects is an actually an array of reference variables
- Because `Strings` are objects, we know that arrays can contain objects.

```
BankAccount[] accounts = new BankAccount[5];
```

- The `accounts` variable holds the address
  - of an `BankAccount` array.
- The array is an array of references to `BankAccount` objects.



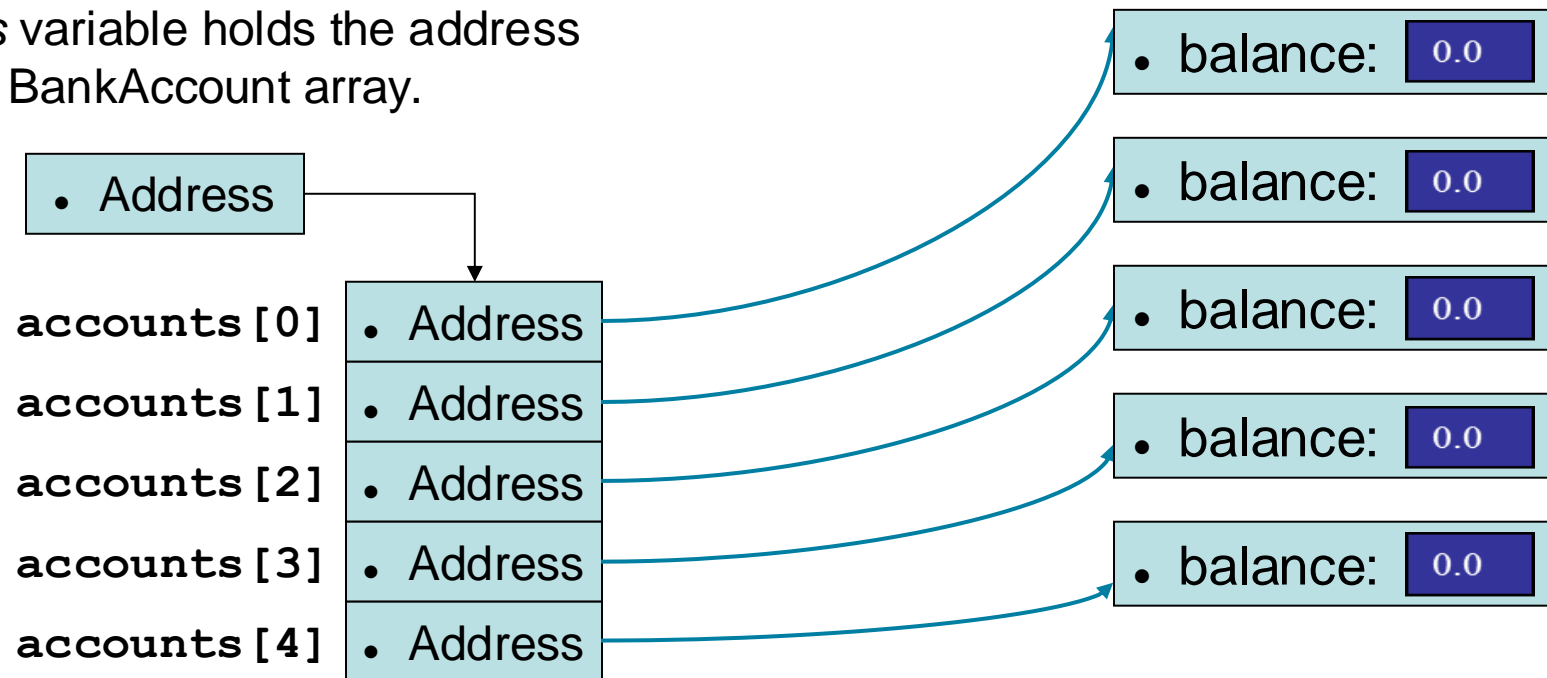


# Array of Objects (2 of 2)

- Each element needs to be initialized.

```
for (int i = 0; i < accounts.length; i++)  
    accounts[i] = new BankAccount();
```

- The *accounts* variable holds the address
  - of an BankAccount array.



# Passing Arrays as Arguments

# Passing Arrays as Arguments (1 of 4)

- When a single element of an array is passed to a method it is handled like any other variable.
- More often you will want to write methods to process array data by passing the entire array, not just one element at a time.

# Passing Arrays as Arguments (2 of 4)

- Arrays are objects.
- Their references can be passed to methods like any other object reference variable.

`showArray(numbers);`

• Address

• 5	• 10	• 15	• 20	• 25	• 30	• 35	• 40
-----	------	------	------	------	------	------	------

```
public static void showArray(int[] array)
{
    for (int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
}
```

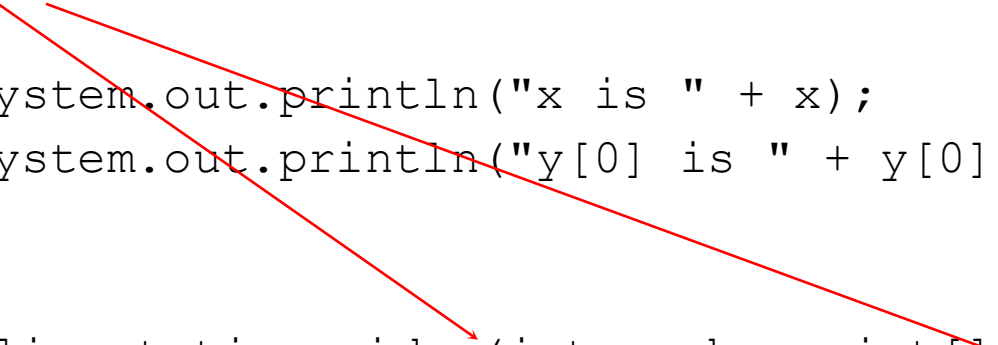
# Passing Arrays as Arguments (3 of 4)

Java uses *pass by value* to pass arguments to a method. There are important differences between passing a value of variables of primitive data types and passing arrays:

- For a parameter of a primitive type value, the actual value is passed.
  - Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.
- For a parameter of an array type, the value of the parameter contains a reference to an array; this reference is passed to the method.
  - Any changes to the array that occur inside the method body Java will affect the original array that was passed as the argument.

# Passing Arrays as Arguments (4 of 4)

```
public class Test {  
    public static void main(String[] args) {  
        int x = 1; // x represents an int value  
        int[] y = new int[10]; // y represents an array of int values  
  
        m(x, y); // Invoke m with arguments x and y  
  
        System.out.println("x is " + x);  
        System.out.println("y[0] is " + y[0]);  
    }  
  
    public static void m(int number, int[] numbers) {  
        number = 1001; // Assign a new value to number  
        numbers[0] = 5555; // Assign a new value to numbers[0]  
    }  
}
```



# JAVA ArrayList Class

# The `ArrayList` Class

- Similar to an array, an `ArrayList` allows object storage
- Unlike an array, an `ArrayList` object:
  - Automatically expands when a new item is added
  - Automatically shrinks when items are removed
- Requires:

```
import java.util.ArrayList;
```



# Creating an ArrayList

```
ArrayList<String> nameList = new ArrayList<String>();
```

ArrayList's  
can only store  
objects.

Notice the word `String` written inside angled brackets `<>`

This specifies that the `ArrayList` can hold `String` objects.

If we try to store any other type of object in this `ArrayList`, **an error will occur because this array is declared for to hold string objects.**

# Using an ArrayList (1 of 7)

To populate the ArrayList, use the `add()` method:

```
nameList.add("James");  
nameList.add("Catherine");
```

To get the current size, call the `size()` method

```
nameList.size(); // returns 2
```

To access items in an ArrayList, use the `get()` method:

```
nameList.get(1); //In this statement 1 is the index of the item to get.
```

## Using an ArrayList (2 of 7)

- The ArrayList class's `toString()` method returns a string representing all items in the ArrayList

```
System.out.println(nameList);
```

This statement yields :

```
[ James, Catherine ]
```

- The ArrayList class's `remove()` method removes designated item from the ArrayList

```
nameList.remove(1); //This statement removes the second item
```

# Using an ArrayList (3 of 7)

- The ArrayList class's `add()` method with one argument adds new items to the end of the ArrayList
- To insert items at a location of choice, use the `add()` method with two arguments:  

```
nameList.add(1, "Mary");
```

 //This statement inserts the String "Mary" at index 1
- To replace an existing item, use the `set()` method:  

```
nameList.set(1, "Becky");
```

 //This statement replaces "Mary" with "Becky"

## Using an `ArrayList` (4 of 7)

- An `ArrayList` has a capacity, which is the number of items it can hold without increasing its size.
- The default capacity of an `ArrayList` is 10 items.
- To designate a different capacity, use a parameterized constructor:

```
ArrayList<String> list = new ArrayList<String>(100);
```

## Using an ArrayList (5 of 7)

- You can store any type of *object* in an ArrayList

```
ArrayList<BankAccount> accountList = new  
ArrayList<BankAccount>();
```

This creates an ArrayList that can hold BankAccount objects.



# Using an ArrayList (6 of 7)

```
// Create an ArrayList to hold BankAccount objects.
ArrayList<BankAccount> list = new ArrayList<BankAccount>();

// create three BankAccount objects and add them to the ArrayList.
list.add(new BankAccount(100.0));
list.add(new BankAccount(500.0));
list.add(new BankAccount(1500.0));

// Display each item.
for (int index = 0; index < list.size(); index++)
{
    BankAccount account = list.get(index);
    System.out.println("Account at index " + index + "\nBalance: " + account.getBalance());
}
```

## Using an ArrayList (7 of 7)

The diamond operator: beginning in Java 7, you can use the `<>` operator for simpler `ArrayList` declarations:

- 

No need to specify the data type here.

```
ArrayList<String> list = new ArrayList<>();
```

- Java infers the type of the `ArrayList` object from the variable declaration.



# Ordered Lists

versus unordered lists

# Book Shelf Example

- How to organize books on your book shelf?

- **Solution 1:**

- Put the books in a random order
    - Whenever you get a new book place it as the last book on the shelf
    - What are the pros and cons of this solution?
      - **Good:** needs a short time to insert a book
      - **Bad:** takes a long time to find a specific book by title

- **Solution 2:**

- Sort the books alphabetically by book title
    - What are the pros and cons of this solution?
      - **Good:** finding a specific title is very fast
      - **Bad:** inserting a new book requires more time in order to shift the books to make a space for the new book.

# Tradeoffs – Which Operation is Done Most Often?

## Unordered Lists

- Adding an element is fast
- Finding an element is slow
- Removing an element is fast



## Ordered Lists

- Adding an element is slow
- Finding an element is faster
- Removing an element is slow



# Invariants

Something that is always true

# Invariants - Always has to be true no matter what.

- An invariant is an explicit statement of the rules dictating how instance variables are to be used
- All methods may assume that the invariant is valid when they are called
- Each method is responsible for ensuring continuing validity of the invariant when the method returns

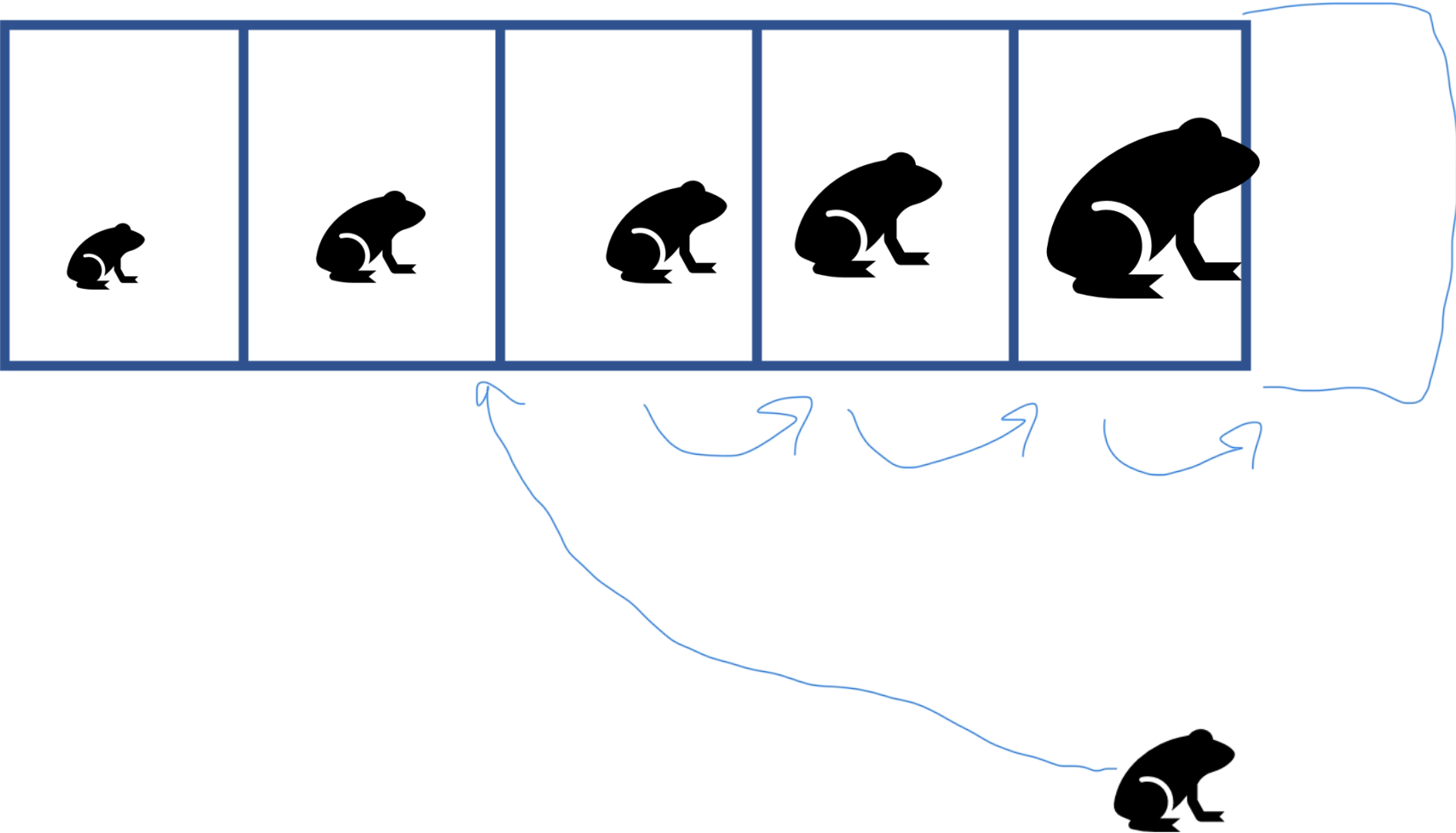
# Invariants of Sorted Array

- The number of items contained in the collection is stored in an instance variable conventionally called `manyItems`
- The ordered list entries are stored in an instance array variable conventionally called `data`, from position `data[0]` to position `data[manyItems-1]`
- Items are stored in order

# Add an item to a sorted array

DO NOT USE A SORTER – Learning how those are implemented

# Adding to array





# Pseudocode for Inserting in Order

Make sure there is room to add it

Starting with the last item in the collection, repeat until not moving over

    If item to add is less than that item, move that item over

Add item to “empty” spot

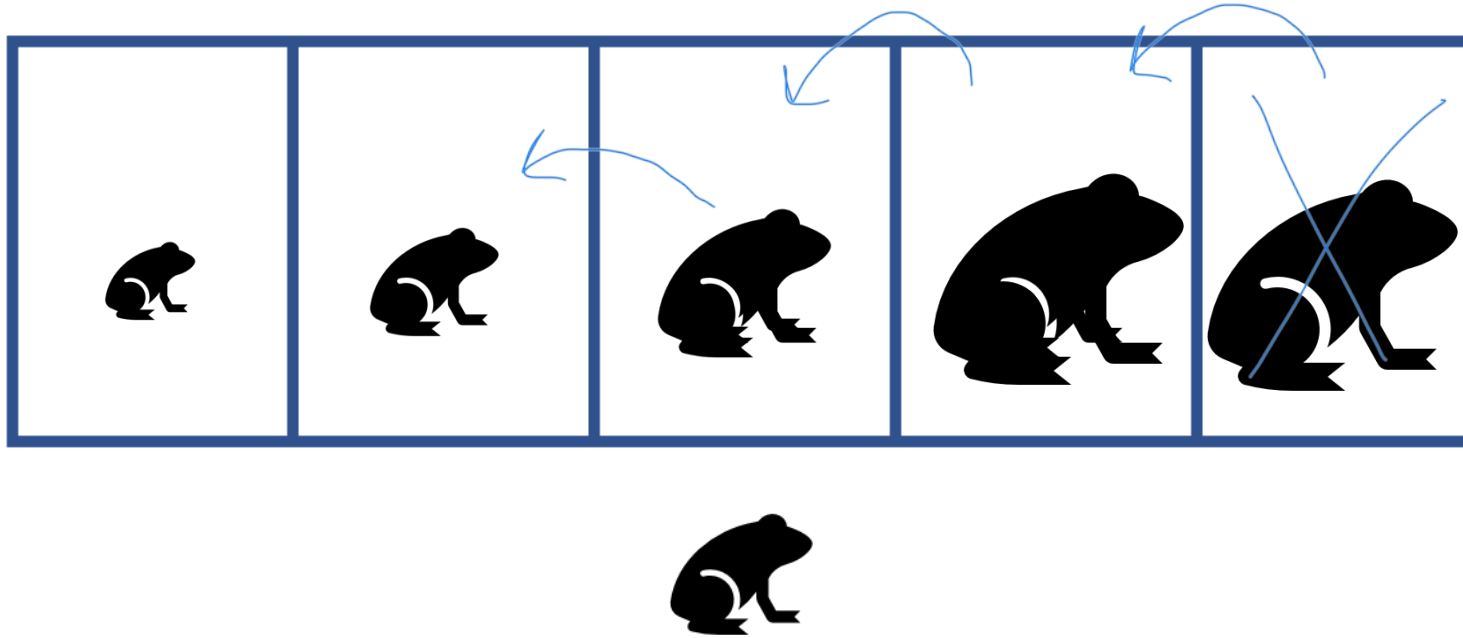
Increase the instance variable manyItems by one

\*\*\*Make sure will work for border cases (first thing added, last thing added)

# Remove an item from a sorted array

DO NOT USE A SORTER!

# Deleting from array



# Pseudocode for Removing an Item from an Ordered List

- Find the item

- Starting at that position until the last item in the array

- Move the item at the next slot over to the current slot

- Decrease manyItems

# Activity

- Update the add methods to the AnimalShelter class in the Animal Shelter project to maintain a sorted collection.
- Create a remove method in the AnimalShelter class that accepts an Animal to remove. This method should return true if the animal was in the collection and has been removed, and false if the animal was not in the collection.
- Test the new methods from the driver. Does the collection stay sorted?