Arrays and ArrayLists

David Greenstein
Monta Vista High School

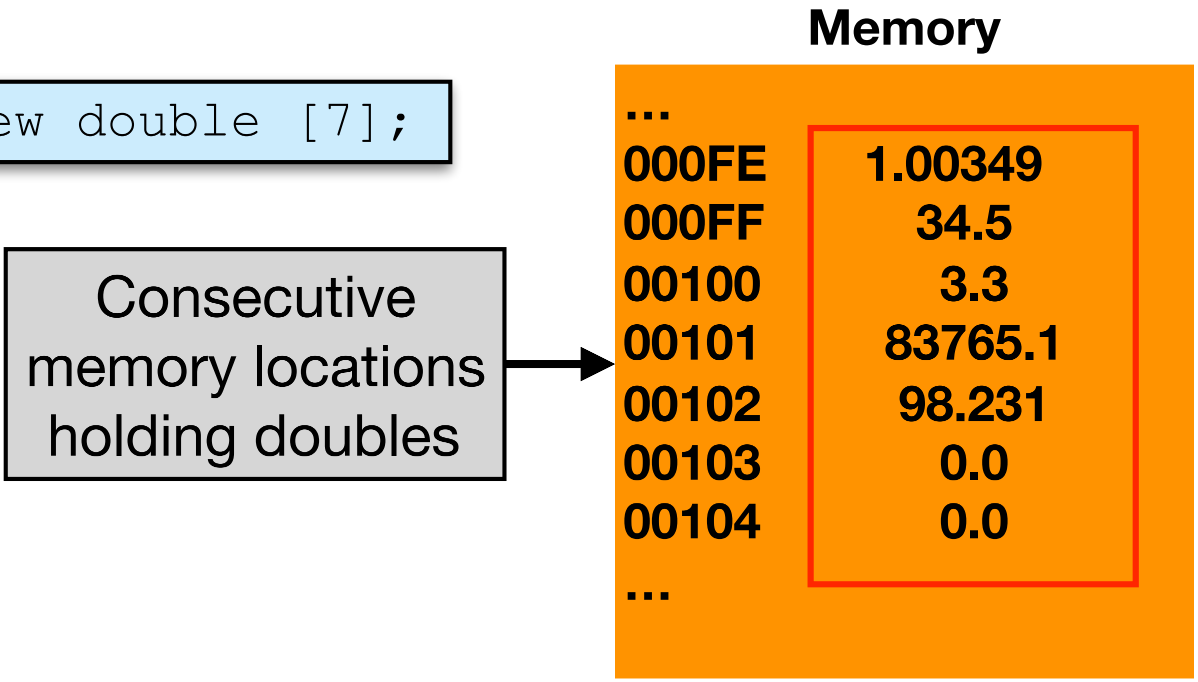
Array

- An **array** is a block of consecutive memory locations that hold values of the same data type.

- Individual locations are called array’s **elements.**

- When we say **“element”** we often mean the value stored in that element.

```
double [] arr = new double [7];
```



Consecutive memory locations holding doubles

Memory

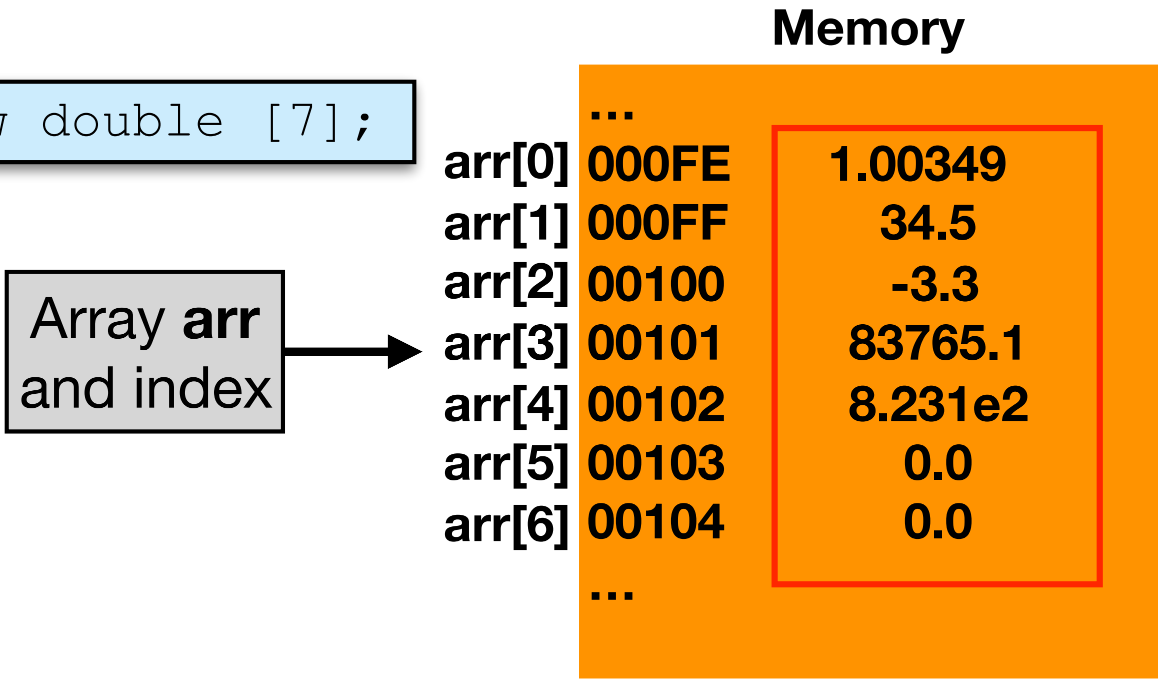
…	
000FE	**1.00349**
000FF	**34.5**
00100	**3.3**
00101	**83765.1**
00102	**98.231**
00103	**0.0**
00104	**0.0**
…	

Array (cont)

- Rather than treating each element as a separate named variable, the whole array gets one name.

- Specific array elements are referred to by using the array's name and the element's number, called the **index** or **subscript**.

```
double [] arr = new double [7];
```



Array **arr** and index

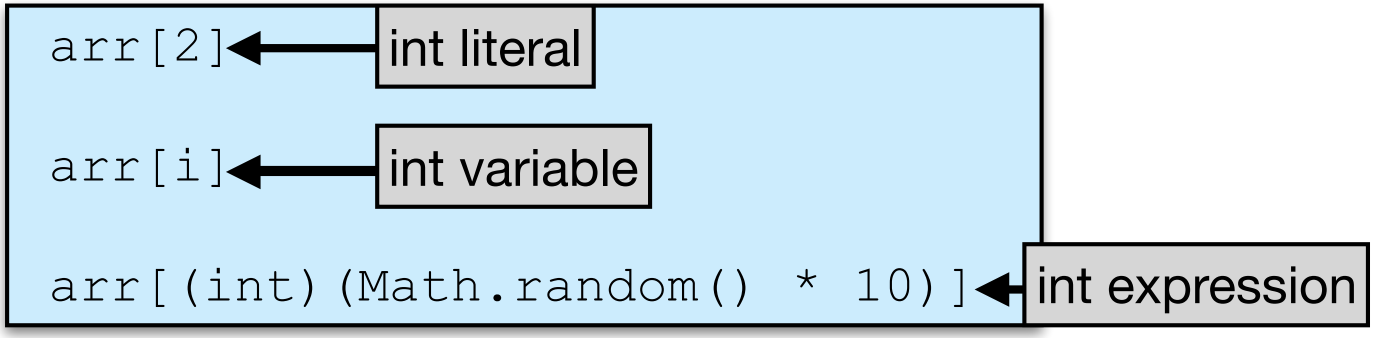
Memory

	...	
arr[0]	**000FE**	**1.00349**
arr[1]	**000FF**	**34.5**
arr[2]	**00100**	**-3.3**
arr[3]	**00101**	**83765.1**
arr[4]	**00102**	**8.231e2**
arr[5]	**00103**	**0.0**
arr[6]	**00104**	**0.0**
	...	

Indices/Subscripts

- **Indices** can be any **int** value represented by a literal number, an int variable, or an expression that evaluates to an **int** value.

- The range of valid indices start at the first element indicated by a zero (0), to the last element indicated by array’s length minus 1 (arr.length - 1).



```
arr[2]
arr[i]
arr[(int)(Math.random() * 10)]
```

int literal

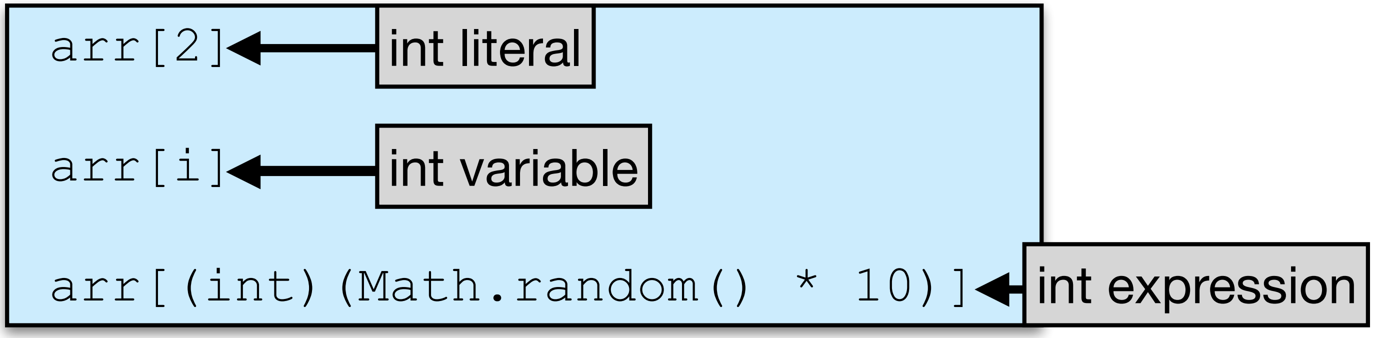
int variable

int expression

Indices/Subscripts (cont)

- In Java, an array is initialized with fixed length that cannot be changed.

- The Java interpreter checks the values of indices at run time and throws **ArrayIndexOutOfBoundsException** if an index is negative or if it is greater than the array length - 1 (eg. `arr.length - 1`).



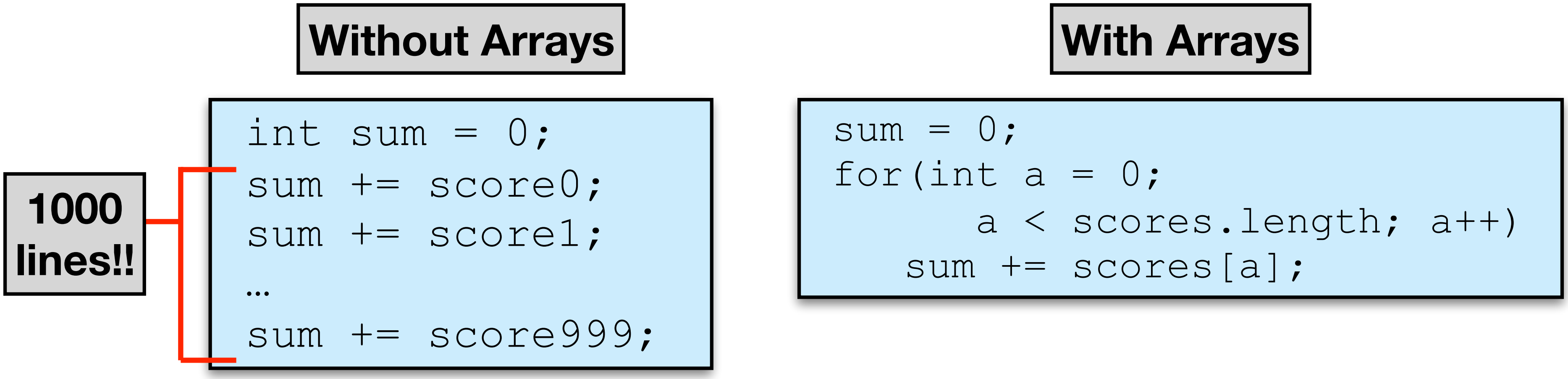
```
arr[2]
arr[i]
arr[(int)(Math.random() * 10)]
```

Power of Arrays

- Arrays allow us to gather similar information together into a list. Most programming languages have arrays with indices.

- Indices give direct access to each element quickly.

- Indices can be computed during runtime to help with repeating similar operations over the list.



Without Arrays

1000 lines!!

```
int sum = 0;
sum += score0;
sum += score1;
…
sum += score999;
```

With Arrays

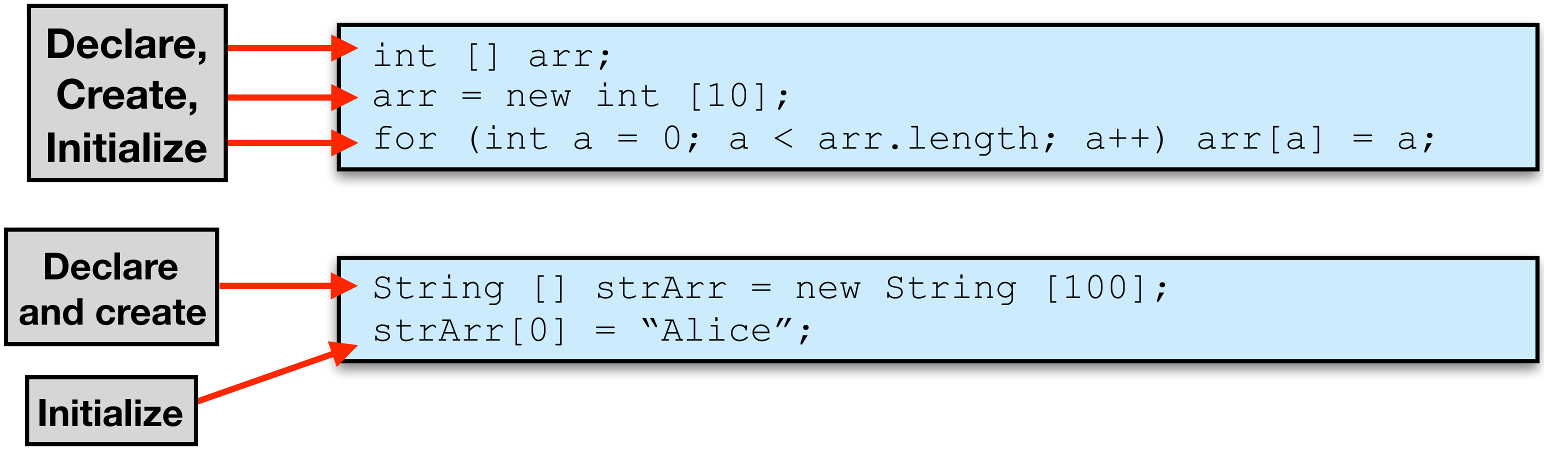
```
sum = 0;
for(int a = 0;
        a < scores.length; a++)
    sum += scores[a];
```

Arrays are Objects

- An array in Java is an object without methods. Therefore, there is no class that describes an array. (Remember, array.**length** is a property!)

- An array is created and memory is allocated using the **new** operator (just like other objects!).

- An array must be declared and created before it can be initialized.



Declare, Create, Initialize

```
int [] arr;
arr = new int [10];
for (int a = 0; a < arr.length; a++) arr[a] = a;
```

Declare and create

Initialize

```
String [] strArr = new String [100];
strArr[0] = “Alice”;
```

Initializing Arrays

- Arrays can contain either primitives or objects.

- Once an array is created it, each element contains the same default values as a field of a class:
  - numbers are zero
  - boolean are false
  - char is the null character
  - objects are null pointers.

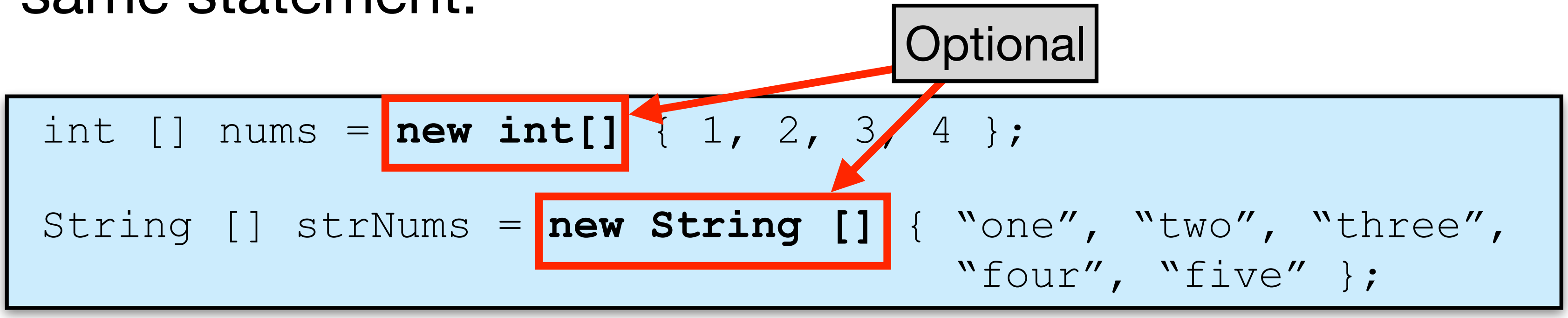
int array contains 0's

String array contains null pointers

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

String [] strArr = new String [100];
```

Initializing Arrays (cont)

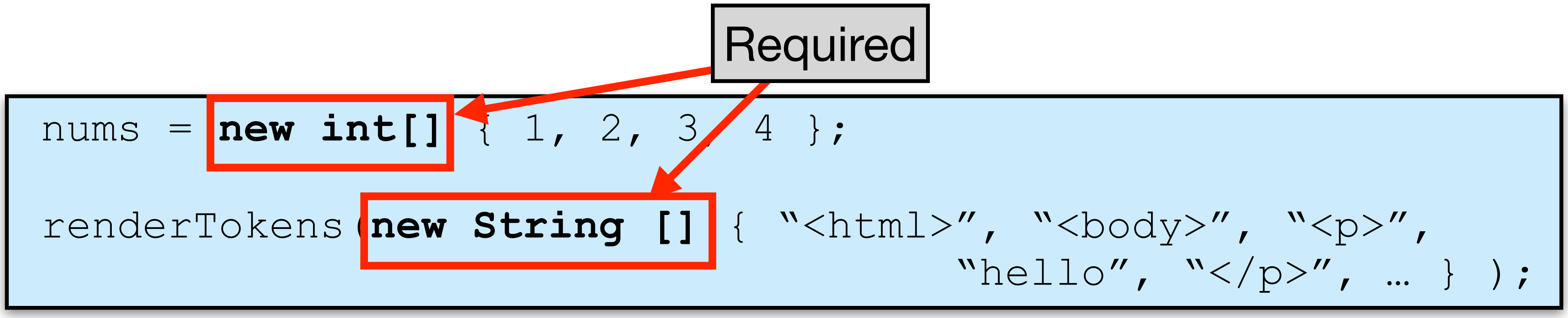
- Arrays can be declared, created, and initialized in the same statement.



Optional

```
int [] nums = new int[] { 1, 2, 3, 4 };

String [] strNums = new String [] { “one”, “two”, “three”,
                                    “four”, “five” };
```

- If you want to create an array on-the-fly, always use “**new** dataType **[ ]**”

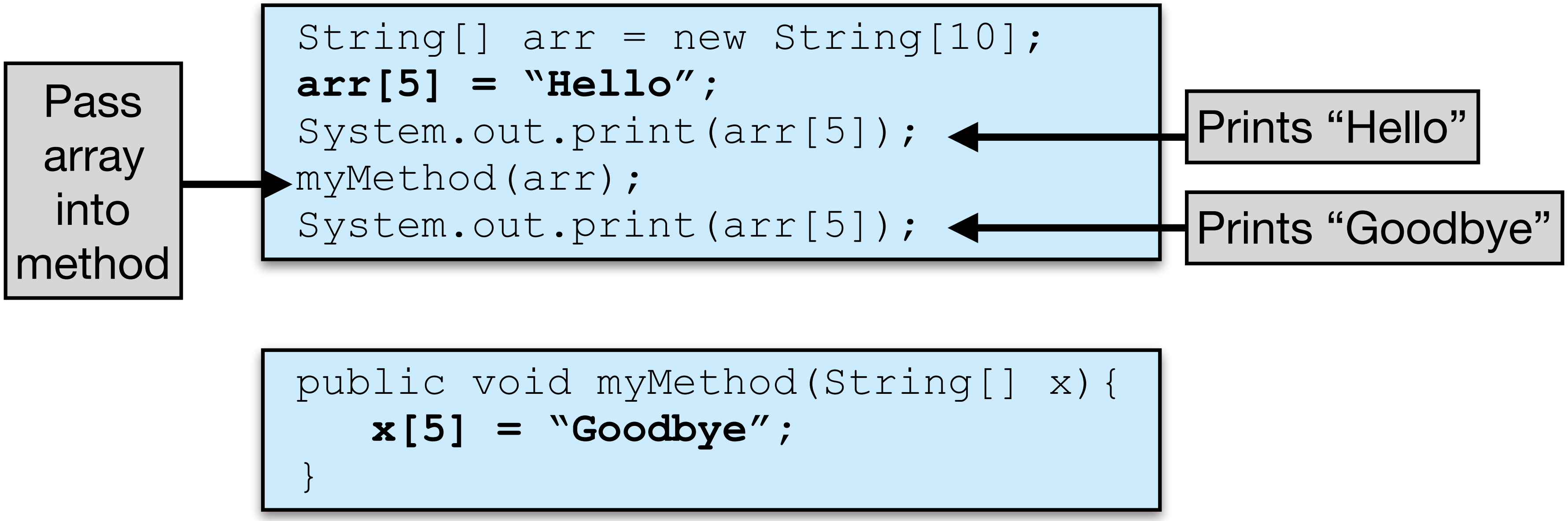


Required

```
nums = new int[] { 1, 2, 3, 4 };

renderTokens (new String [] { “<html>”, “<body>”, “<p>”,
                                    “hello”, “</p>”, … } );
```

Arrays are Mutable

- You can change an element in an array and all references to the array are also changed.



Pass array into method

```
String[] arr = new String[10];
arr[5] = “Hello”;
System.out.print(arr[5]);
myMethod(arr);
System.out.print(arr[5]);
```

Prints “Hello”

Prints “Goodbye”

```
public void myMethod(String[] x){
    x[5] = “Goodbye”;
}
```

Array Length

- The **length** of an array is determined when it is created, either by the contents of braces {} or a number explicitly in brackets. (ie. [10])

```
char[] letters = { ‘a’, ‘m’, ’s’, ‘z’ };
String[] names = new String[10];
```

- In Java, the **length** of an array is a property and not a method.

```
char[] arr = new char[10];
int len = arr.length;
```

Correct

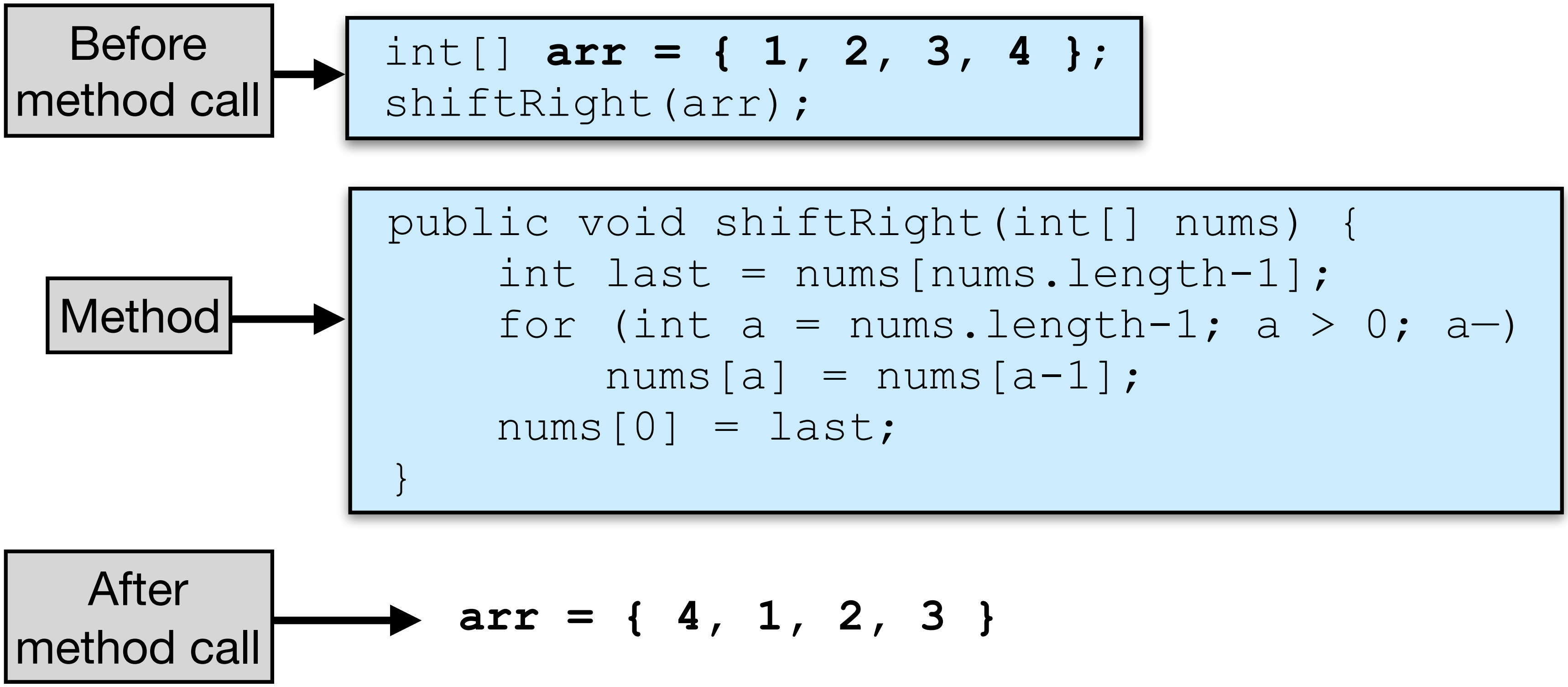
```
char[] arr = new char[10];
int len = arr.length();
```

Syntax Error!!

Passing Arrays to Methods

- As other objects, an array is passed to a method as a reference. (pass-by-reference)

- The **elements** of the original array are not copied and are accessible in the method’s code.



Before method call

```
int[] arr = { 1, 2, 3, 4 };
shiftRight(arr);
```

Method

```
public void shiftRight(int[] nums) {
    int last = nums[nums.length-1];
    for (int a = nums.length-1; a > 0; a—)
        nums[a] = nums[a-1];
    nums[0] = last;
}
```

After method call

```
arr = { 4, 1, 2, 3 }
```

Returning Arrays from Methods

- Sometimes you want a method to return a **new array**.

```
/* Calculate the midpoint between two points.
 * Return a coordinate pair in an array.
 */
public double[] midpoint(double x1, double y1,
                         double x2, double y2) {
   return new double[] { (x1 + x2)/2, (y1 + y2)/2 };
}
```

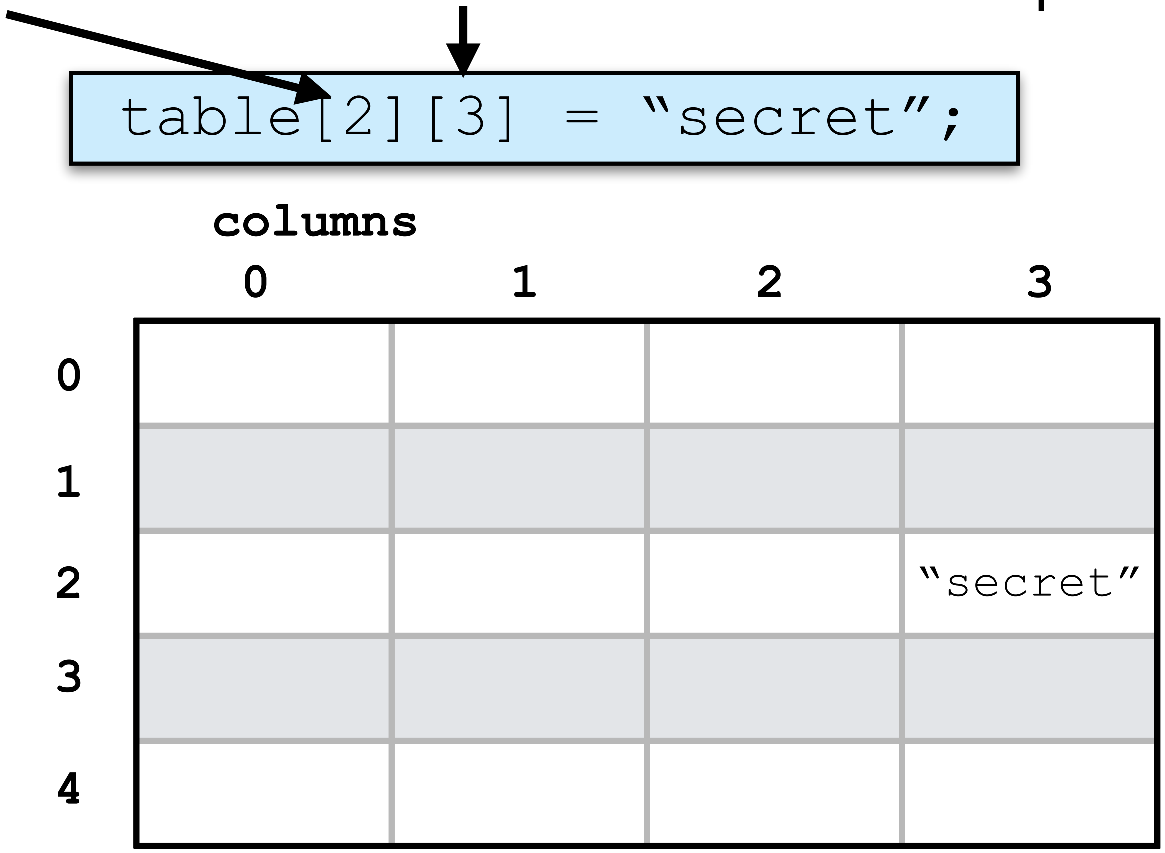
- Sometimes you want to use an array as a parameter, return a new array, but keep the original untouched.

```
/* Returns a copy of an int array. */
public int[] copyInt(int[] inArr) {
   int[] result = new int[inArr.length];
   for (int a = 0; a < inArr.length; a++)
       result[a] = inArr[a];
   return result;
}
```

Two-dimensional Arrays

- **Two-dimensional arrays** allow us to represent 2D figures like tables, grids, game boards, images, etc.

- An element in a two-dimensional array is accessed using a **row index** and **column index.** For example:



```
table[2][3] = “secret”;
```

columns

rows

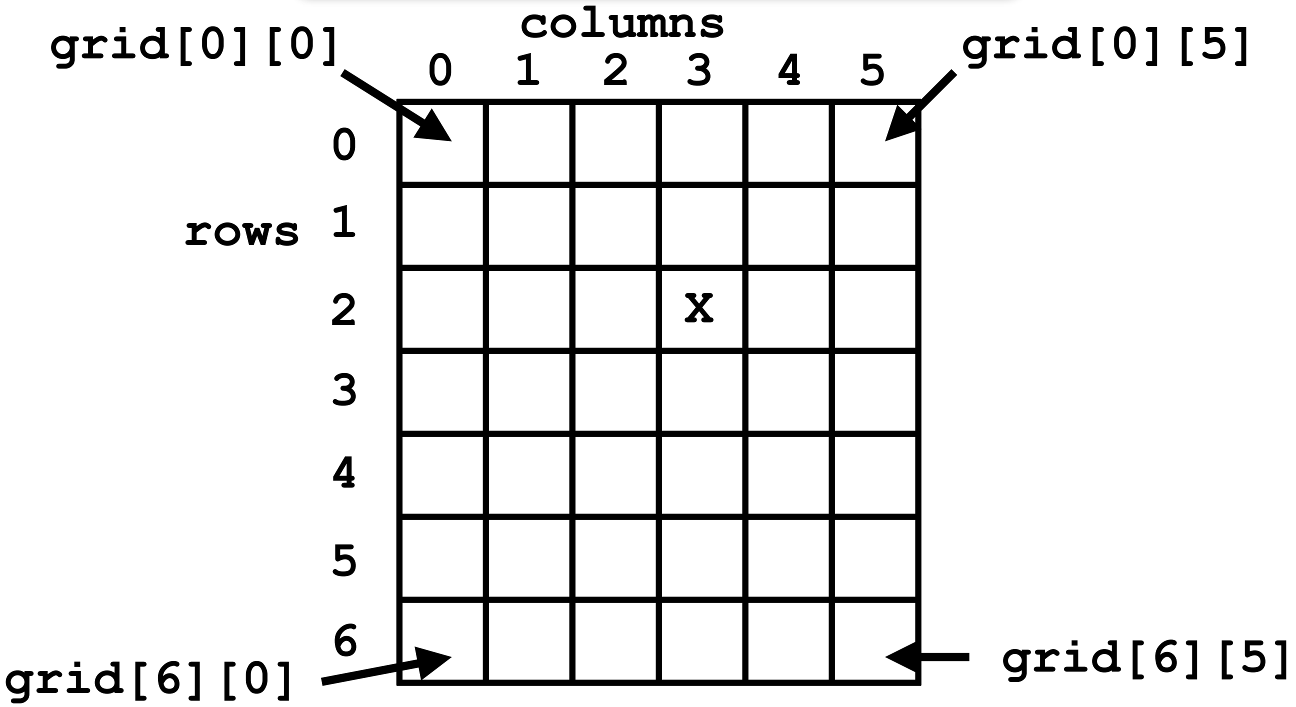
	0	1	2	3
0				
1				
2				“secret”
3				
4				

Two-dimensional Arrays

- **Two-dimensional arrays** allow us to represent 2D figures like tables, grids, game boards, images, etc.

- An element in a two-dimensional array is accessed using a **row index** and **column index.** For example:

```
grid[2][3] = 'X';
```



columns

0 1 2 3 4 5

rows

0 1 2 3 4 5 6

X

grid[0][0]

grid[0][5]

grid[6][0]

grid[6][5]

Declaring 2-D Arrays

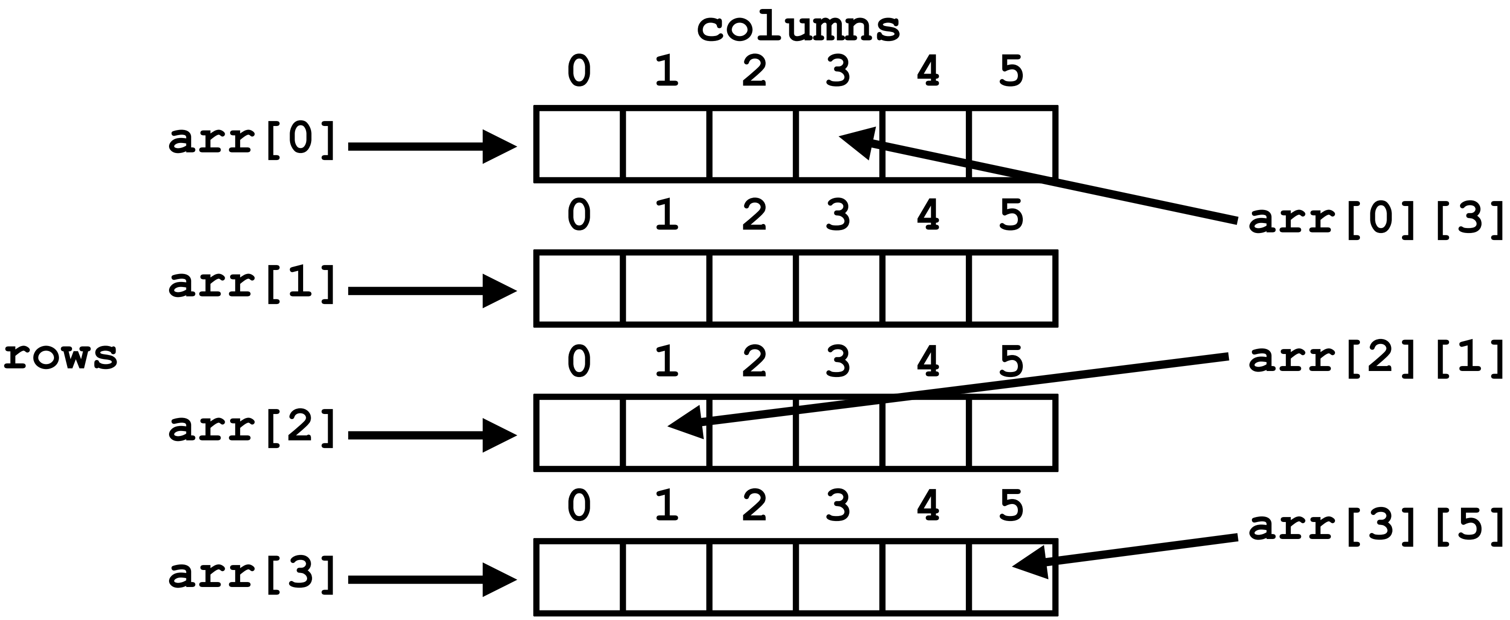
```
// array with 5 rows and 7 columns
double[][] arr = new double [5][7];

// 2D array containing objects
Color [][] pixels = new Color[480][720];

// Declaring and initializing 2D array
int [][] matrix = { { 1, 2, 3 },
                    { 4, 5, 6 },
                    { 7, 8, 9 } };
```

2D Array Dimensions

- A **two-dimensional array** is really a 1-D array of 1-D arrays.

```
int[][] arr = new int[2][3];
```

- **arr[k]** is a 1-D array in the k-th row.
- **arr.length** is the number of rows.
- **arr[k].length** is the number of columns in row k.

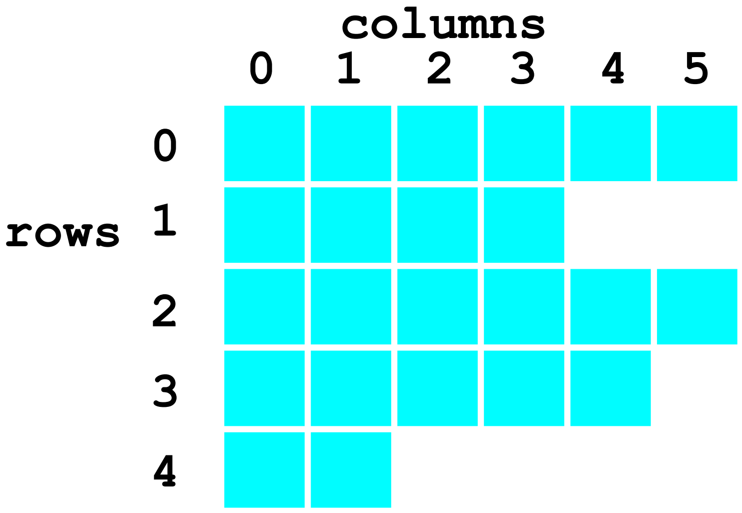


2D Array Dimensions (cont)

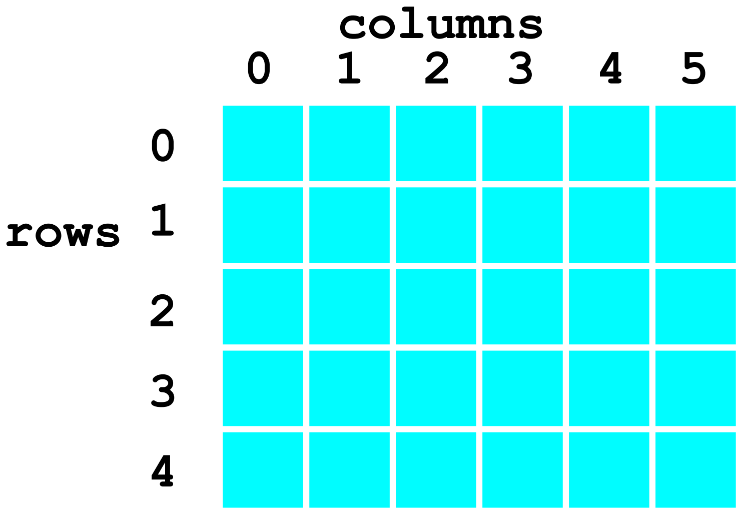
- Java allows **“jagged” arrays** in which different rows have a different number of columns. (also called “ragged” array)

- In a rectangular array **m[0].length** is the number of columns in all rows.

“Jagged” array



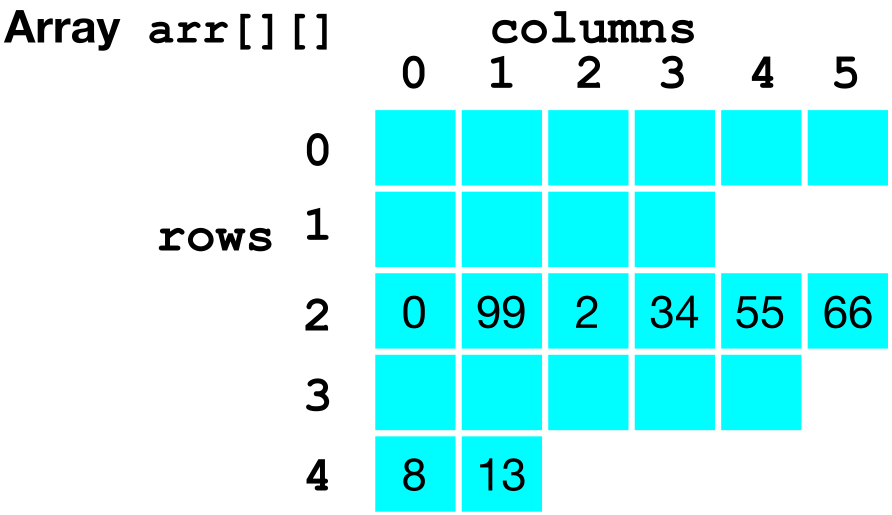
Rectangular array



2D Array Dimensions (cont)

- Creating and initializing **“jagged” arrays** is similar to 1D arrays.

```
int[][] arr = new int[5][0];
arr[0] = new int[6];
arr[1] = new int[4];
arr[2] = new int[] { 0, 99, 2, 34, 55, 66 };
arr[3] = new int[5];
arr[4] = new int[] { 8, 13 };
```



Array `arr[][]`

		columns					
		0	1	2	3	4	5
	0						
rows	1						
	2	0	99	2	34	55	66
	3						
	4	8	13				

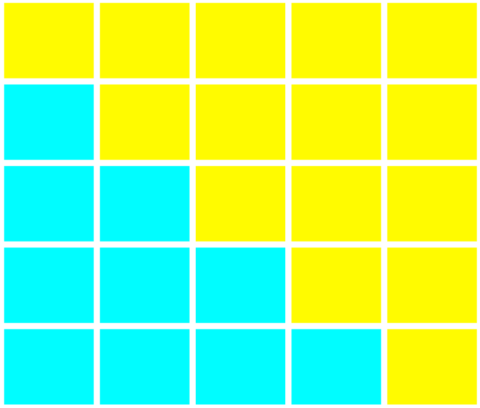
2-D Arrays and Nested Loops

- To reach each element in a rectangular 2D array it is necessary to use nested loops.

```
for (int r = 0; r < arr.length; r++)
   for (int c = 0; c < arr[0].length; c++) {
       // process arr[r][c]
   }
```

- **“Triangular” loops** are nested loops that use the row’s value to determine the column’s range of values.

```
// transpose a square matrix
for (int r = 1; r < arr.length; r++)
   for (int c = 0; c < r; c++) {
      double temp = arr[r][c];
      arr[r][c] = arr[c][r];
      arr[c][r] = temp;
   }
```



“For Each” Loop

- Introduced in Java version 5. (Lab computers are on version 7; the current version is 11)

- One-dimensional array example:

```
double[] oneDArr = new double[10];
…
for (double element : oneDArr) {
   // process element
}
```

- Two-dimensional array example:

```
String[][] twoDArr = new String[32][24];
…
for (String[] strArr : twoDArr)
   for (String element : strArr) {
   // process element
   }
```

“For Each” Loop (cont)

- **Best** for doing identical processes on each element regardless of their position in the array.

```
// get sum of elements in array
int sum = 0;
for (int element : anyArr)
     sum += element;
```

- **Not good** when you are doing operations for specific indices.

```
// get sum of every other element
int sum = 0;
int cnt = 0;
for (int element : anyArr) {
     if (cnt % 2 == 0) sum += element;
     cnt++;
}
```

Better for loop

```
// get sum of every other element
int sum = 0;
for (int a = 0; a < anyArr.length; a++)
     if (a % 2 == 0) sum += anyArr[a];
```

ArrayLists

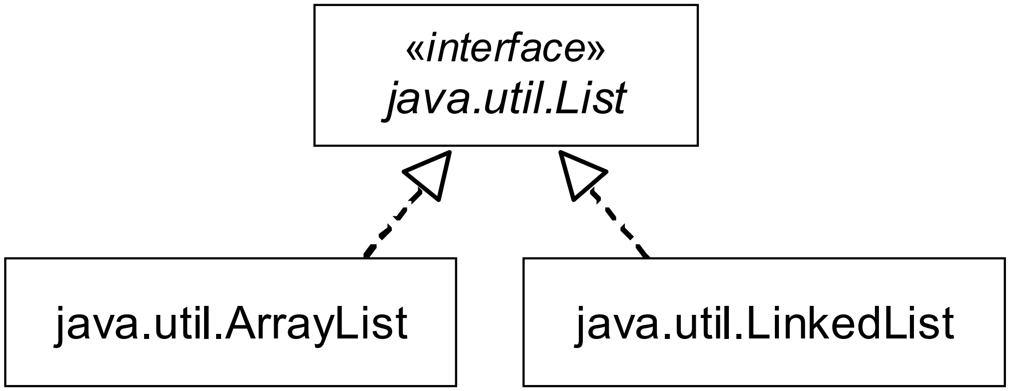
java.util.ArrayList<E>

- **ArrayList** is a class in the **java.util** package.

```
java.util.ArrayList<E>
```

- The **<E>** stands for “generic data type **E**lement” that will be in the **ArrayList.**
  - For example: **ArrayList<String>** means the **ArrayList** contains **String** elements.

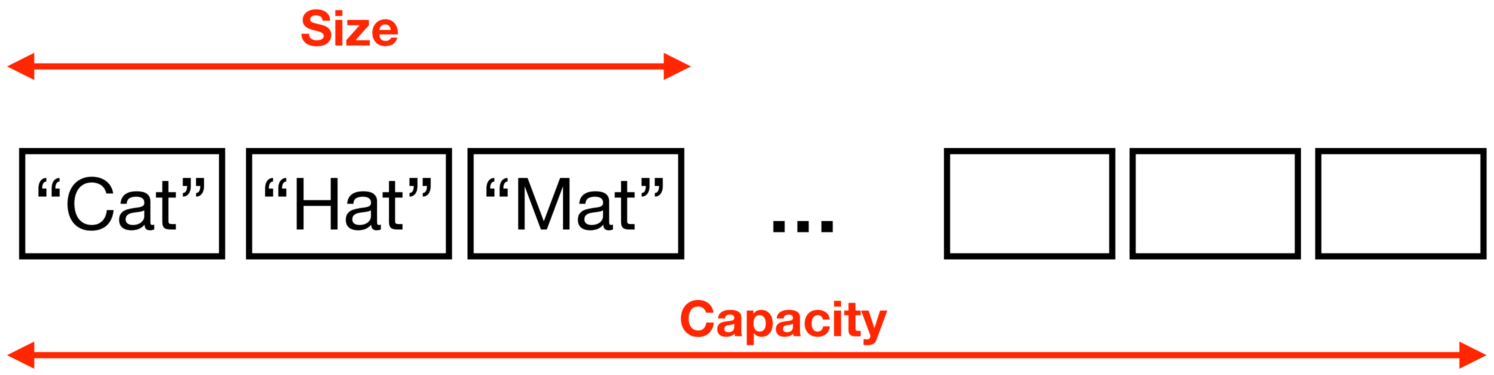
- **ArrayList** implements **java.util.List<E>** interface. (Chapter 19)



java.util.ArrayList<E> (cont)

- **ArrayList** can hold only objects of a specified type <E>, but never primitive data types.

- **ArrayList** keeps track of the number of elements (called **size**).
  - In Java, an **ArrayList** starts with a default capacity of 10 elements.
  - Although **ArrayList** keeps track of the capacity of the list, Java does not share how it grows and shrinks with the programmer.

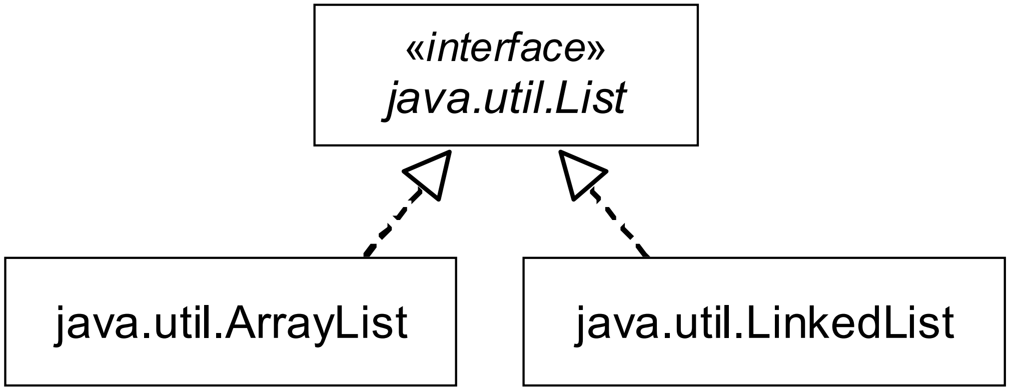


ArrayList and Generics

- Starting with Java version 5, “collection” classes like **ArrayList** began holding objects of a specified data type.
  - We use version 7 in the labs. The current version is 11.

- A “generic” class, like **ArrayList**, forces it to hold only one data type so type checking can be done by the compiler.

```
ArrayList<Integer> nums = new ArrayList<Integer>();

List<String> words = new ArrayList<String>();
```



«interface»
java.util.List

java.util.ArrayList

java.util.LinkedList

ArrayList and Generics (cont)

- A common problem when using generics in your code is the compiler “unchecked or unsafe operations” error.

- Suppose our code looked like this:

No data type specified

```
public class Snake extends ArrayList {
   …
}
```

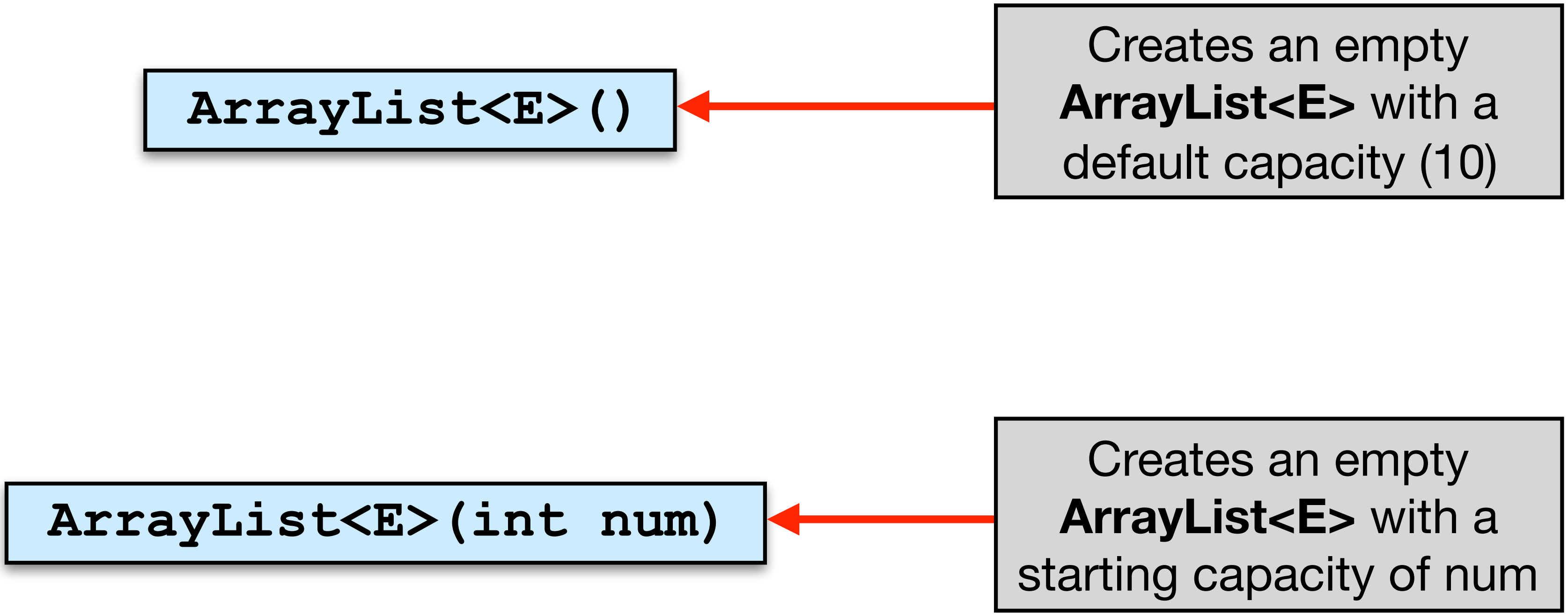
```
% javac Snake.java
Note: Snake.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Fix

```
public class Snake extends ArrayList<Coordinate> {
   …
}
```

ArrayList Constructors

- **ArrayList** has two constructors.



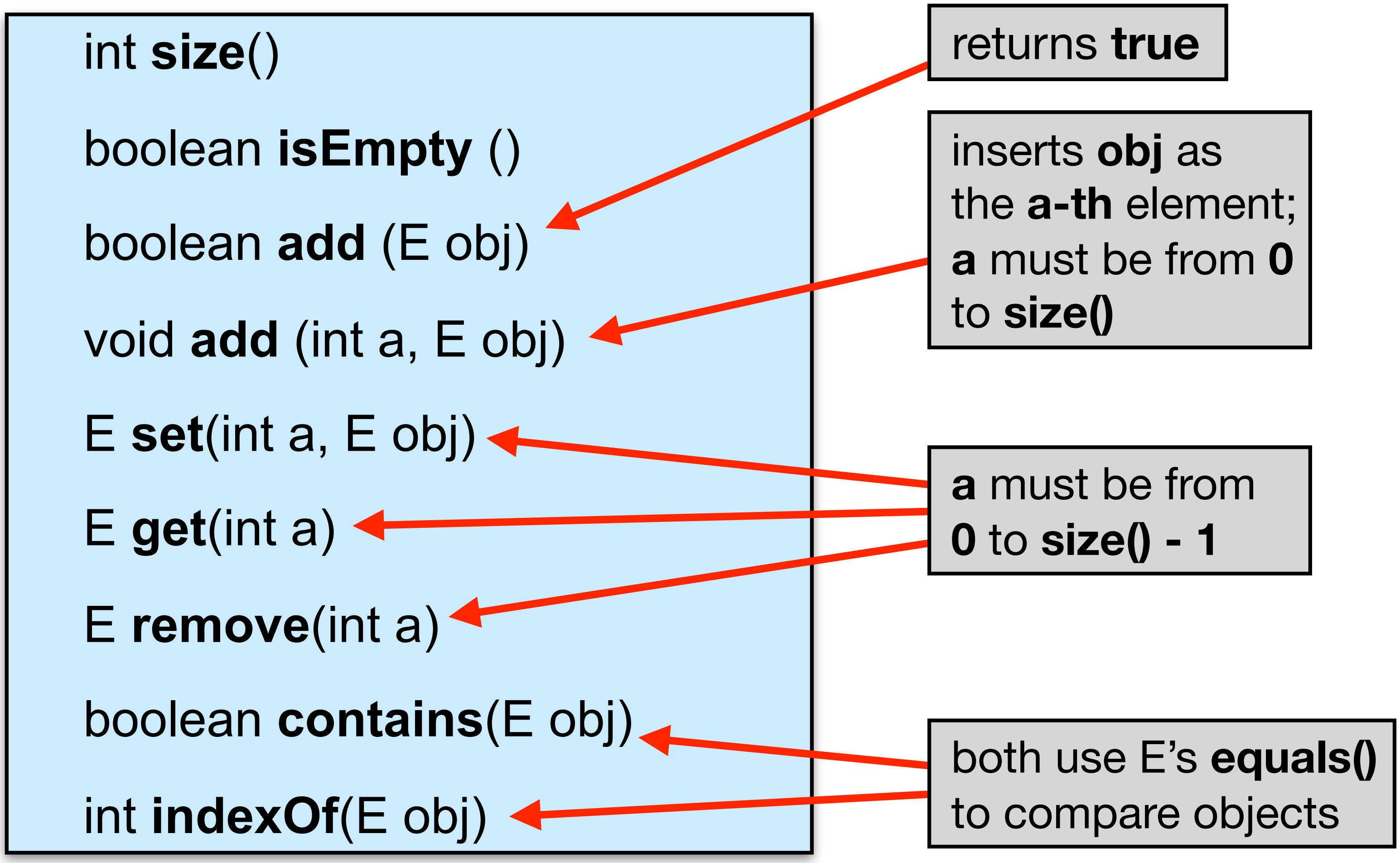
`ArrayList<E>()`

Creates an empty **ArrayList<E>** with a default capacity (10)

`ArrayList<E>(int num)`

Creates an empty **ArrayList<E>** with a starting capacity of num

ArrayList Methods (abridged)



int **size**()

boolean **isEmpty** ()

boolean **add** (E obj)

void **add** (int a, E obj)

E **set**(int a, E obj)

E **get**(int a)

E **remove**(int a)

boolean **contains**(E obj)

int **indexOf**(E obj)

returns **true**

inserts **obj** as the **a-th** element; **a** must be from **0** to **size()**

a must be from **0** to **size() - 1**

both use E's **equals()** to compare objects

ArrayList<E> Details

- **ArrayList** automatically doubles its capacity when it needs more space.

- **get(int a)** and **set(int a, E obj)** are efficient because an array provides random access to its elements.

- It throws an **IndexOutOfBoundsException** when the index is less than 0 or equals **size()** or greater.
  - Therefore, index **a** must be in the range **0 <= a < size()**.
  - In the case of **add(int a, E obj)**, index **a** must be in the range **0 <= a <= size()**.

ArrayList<E> Autoboxing

- Normally, if you have primitives to add to an **ArrayList** you must use a wrapper class, like **Integer** or **Double**.

```
ArrayList<Integer> intNums = new ArrayList<Integer>();
intNums.add(new Integer(5)); // add Integer(5) to list
```

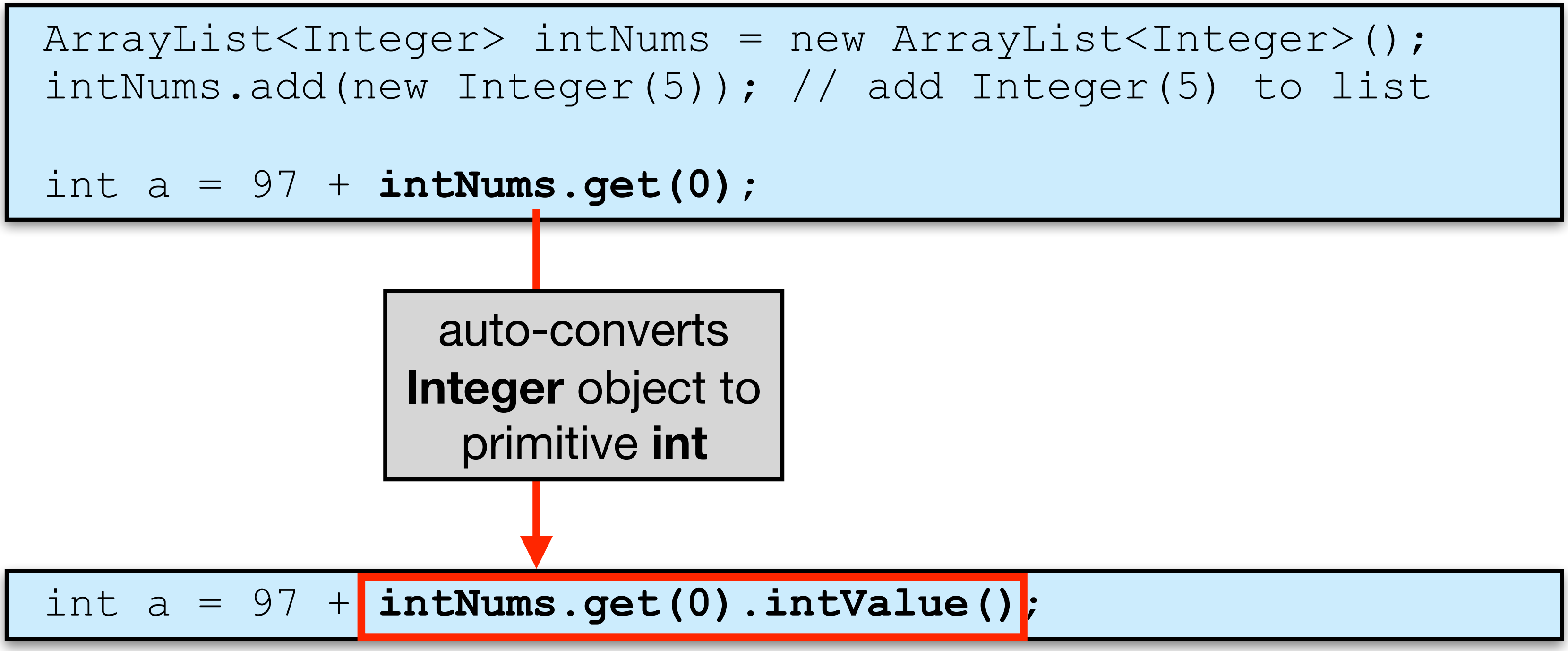
- Since Java version 5, conversion from primitive to wrapper class object is automatic.
  - For example, **int 5** is converted into a **new Integer(5)**.

```
ArrayList<Integer> intNums = new ArrayList<Integer>();
intNums.add(5);                    // add Integer(5) to list
```

- This automatic conversion is called “**autoboxing**” or “**autowrapping**”.

ArrayList<E> Autounboxing

- Since Java 5, it also supports the opposite of autoboxing, called **autounboxing**.



```
ArrayList<Integer> intNums = new ArrayList<Integer>();
intNums.add(new Integer(5)); // add Integer(5) to list

int a = 97 + intNums.get(0);
```

auto-converts **Integer** object to primitive **int**

```
int a = 97 + intNums.get(0).intValue();
```

ArrayList Blunders

```
// remove all “Hello” strings??
for (int a = 0; a < words.size(); a++)
   if (“Hello”.equals(words.get(a)))
       words.remove(a);
```

Only removes one of two consecutive “Hello”s

[“Hello”,”Hello”,”Goodbye”] → [“Hello”,”Goodbye”]

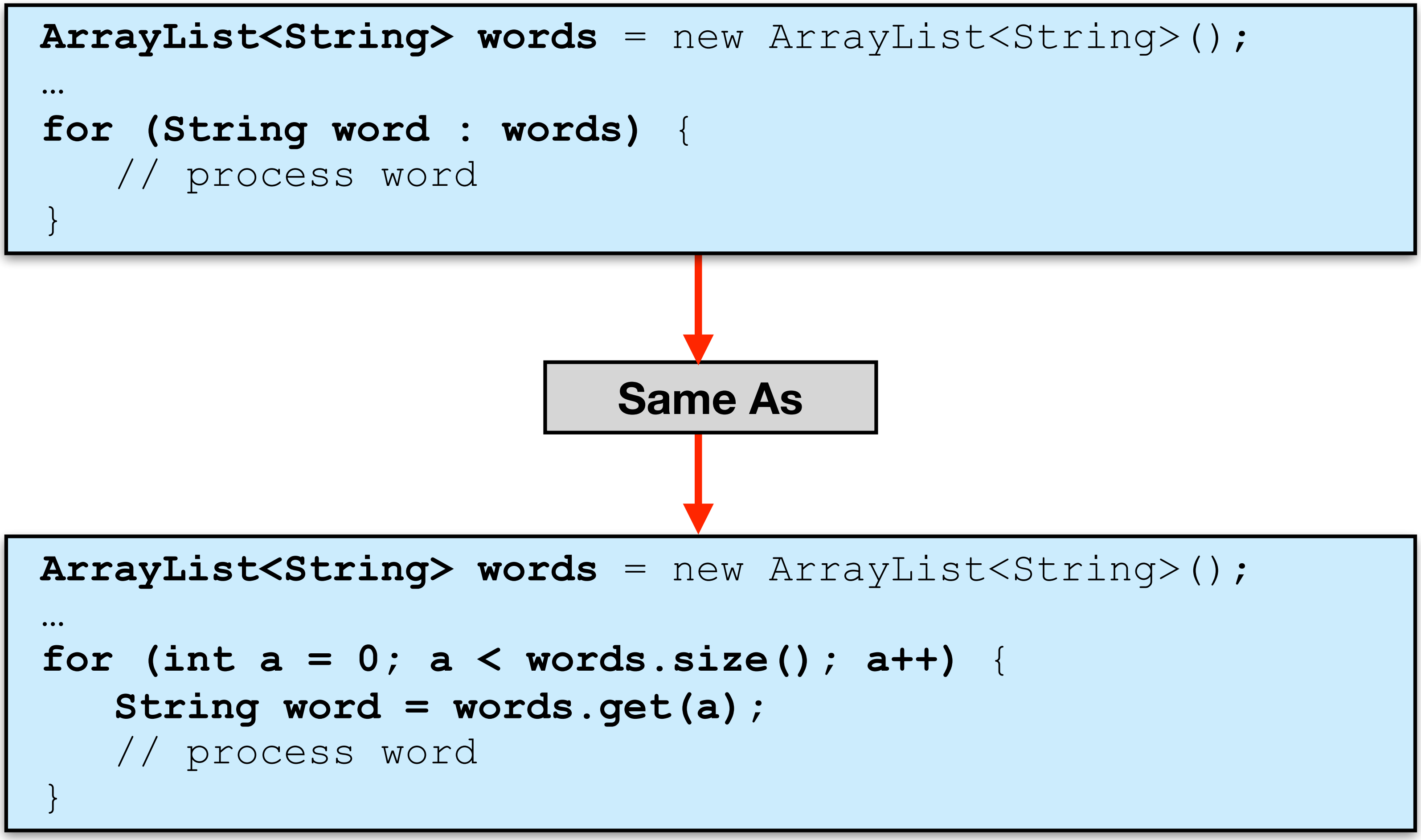
```
// remove all “Hello” strings
for (int a = 0; a < words.size(); a++)
   if (“Hello”.equals(words.get(a)) {
       words.remove(a);
       a--;
   }
```

```
// remove all “Hello” strings
int a = 0;
while (a < words.size())
   if (“Hello”.equals(words.get(a))
       words.remove(a);
   else
       a++;
```

Removes all “Hello”s in list correctly

ArrayList “For Each” Loop

- **For each** works with **Lists** including **ArrayLists**



```
ArrayList<String> words = new ArrayList<String>();
…
for (String word : words) {
    // process word
}
```

Same As

```
ArrayList<String> words = new ArrayList<String>();
…
for (int a = 0; a < words.size(); a++) {
    String word = words.get(a);
    // process word
}
```

Questions?