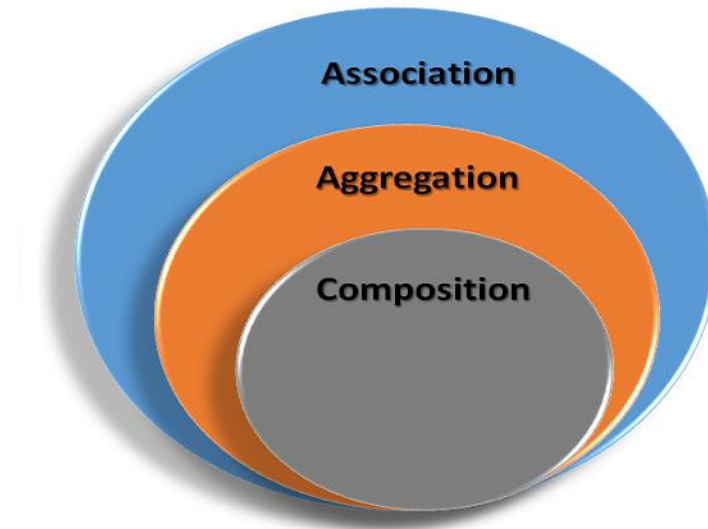# CMPS 251

Relations between Classes
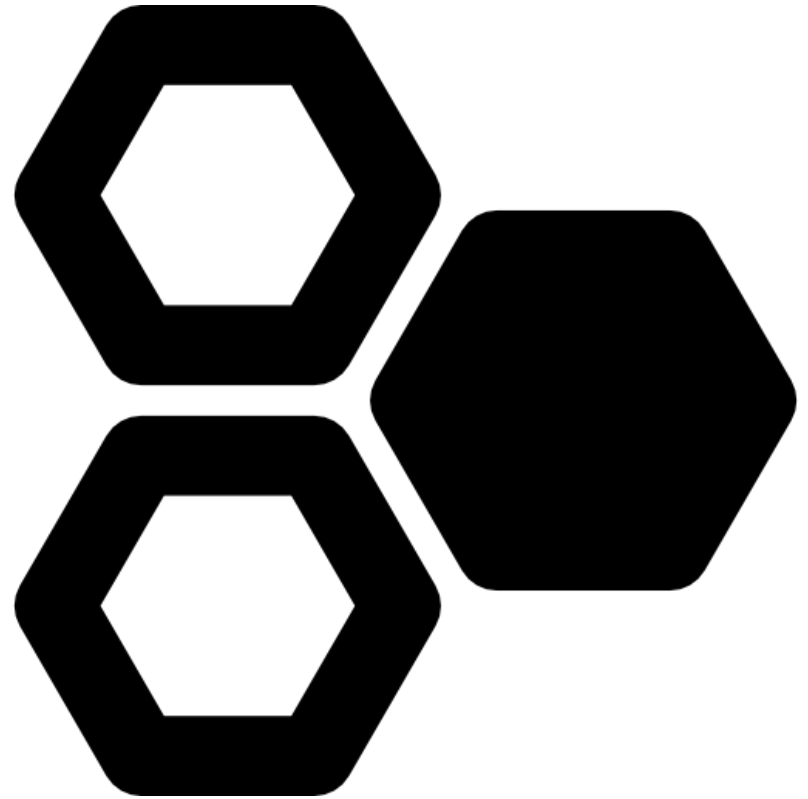

Dr. Abdelkarim Erradi

CSE@QU

Outline

- **Relations between Classes**
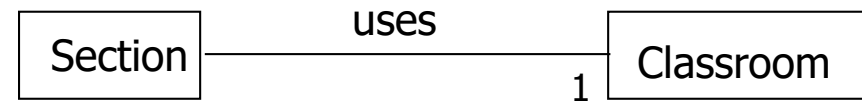
- **Arrays and Lists**

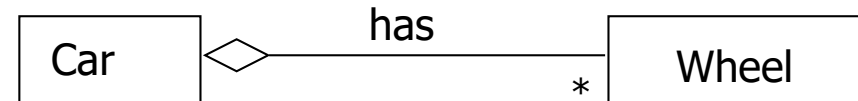# **Relations between Classes**

Relations between Classes

- Classes can be related to other classes in 4 ways:

- **Association** (**uses** without ownership)

Section ── uses ──1── Classroom

- **Aggregation** (**has-a** + Whole-Part relationship)

Car ──◇── has ──*── Wheel

- **Composition** (**has-a** + Part cannot exist without the Whole)

College ──◆── has ──*── Department
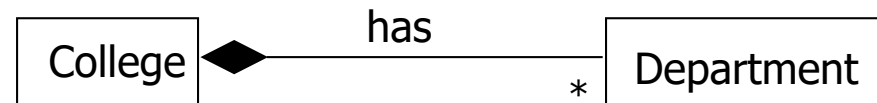
- **Inheritance** (**is-a** relation)

Bank Account
↑
Savings Account   Credit Card Account
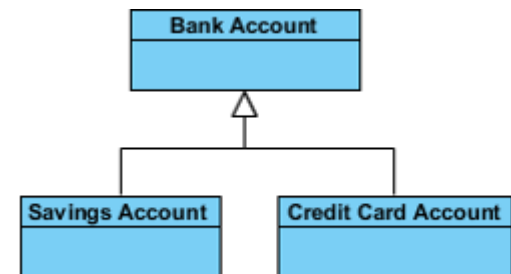
Association

- Association is a very generic relationship used when one class **uses** the functionalities provided by another class

- **No ownership** between the objects and both have their own lifecycle. Both can be created and deleted independently

Student				Instructor
	1..*	learns from		
	teaches		1..*	

Aggregation vs. Composition

- **Aggregation** = WHOLE-PART relationship. PART can exist without the WHOLE.
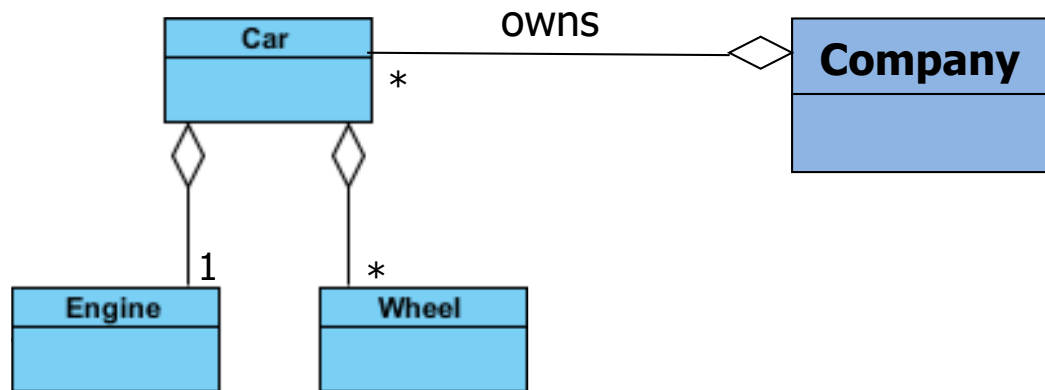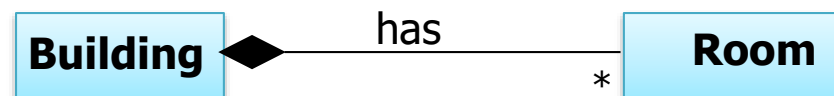

- **Composition** = WHOLE-PART relationship. PART cannot meaningfully exist without the WHOLE
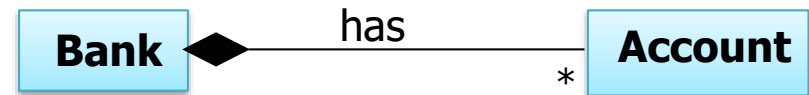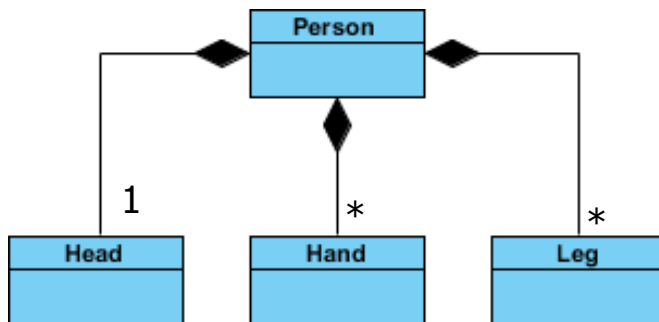
Implementation of bidirectional association

Bank ◆—— 1   **has** ▶   1..*   **Account**

◀ **has**

An association is represented as an attribute in the class

```
class Bank
{
    List<Account> accounts;
    . . . . .
}

class Account
{
    Bank bank;
    . . . . .
}
```

Bank **has many** accounts hence accounts is a list

- The **1 to many** aggregation is represented in the Whole class using an **array** or a **List** attribute

- Part instances can be added to and removed from the list
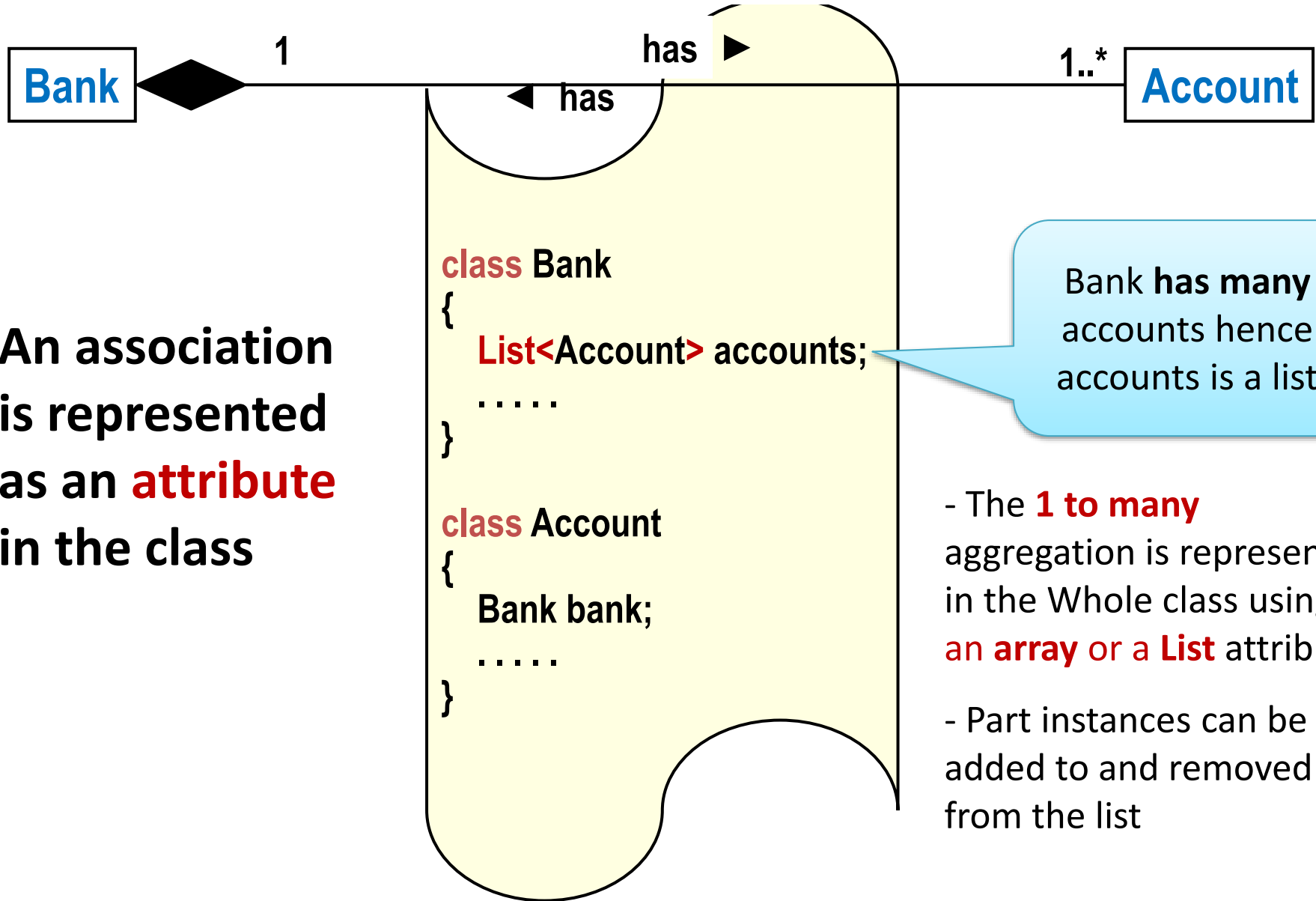
```java
public class Car {
    private Engine engine;

    private List<Wheel> wheels;

    public Car(Engine engine){
        this.engine  = engine;
        this.wheels = new ArrayList<>();
    }

    public addWheel(Wheel wheel){
        wheels.add(wheel);
    }
}

class Engine {
    private String type;
}

class Wheel {
    private int size;
}
```
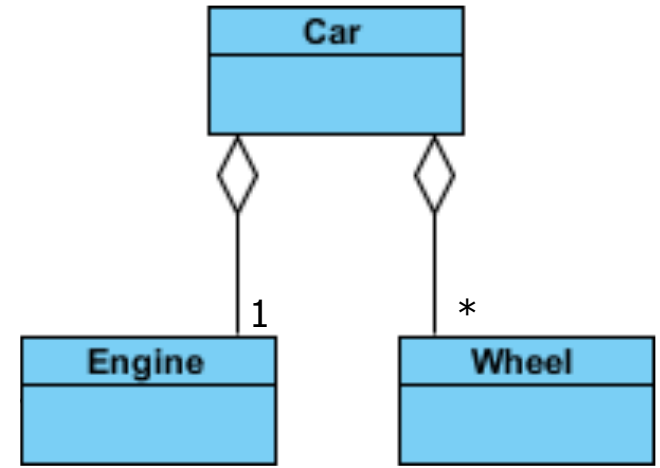
Association vs. Aggregation vs. Composition

- A relationship between two classes is referred as an **Association**
- **Aggregation** is a special form of Association
- **Composition** is a strong form of Aggregation
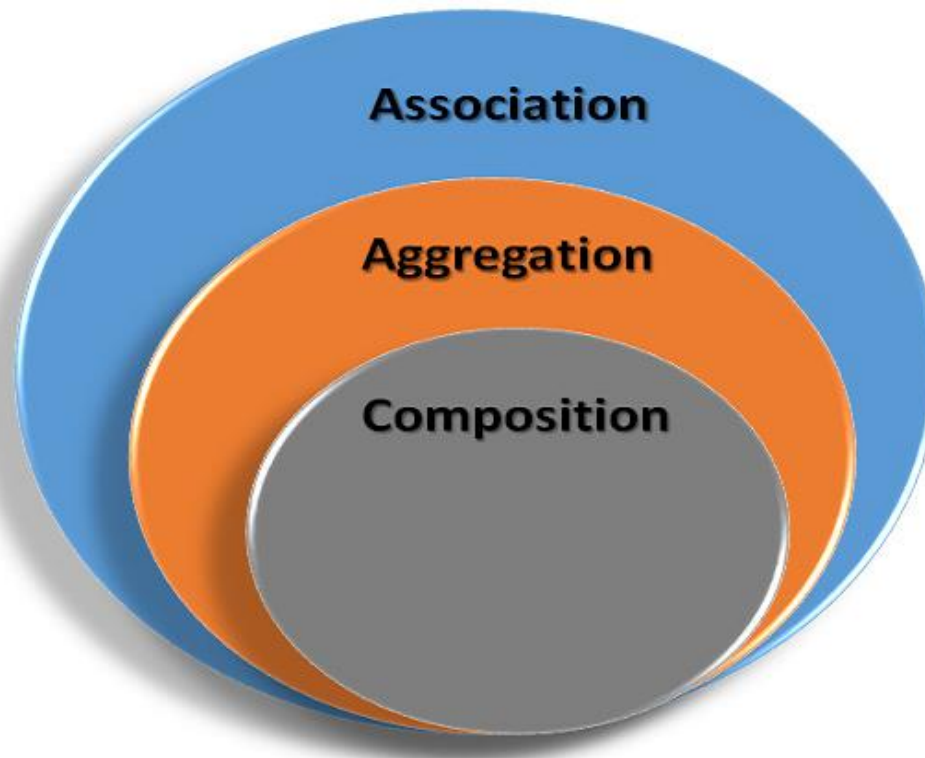
Arrays and Lists

A simple variable stores a single value

MEMORY

int num1 = 10;

int num2 = 20;

int num3 = 30;

num1	10
num2	20
num3	30

An array object stores multiple values of the same type

MEMORY

```
int[] num = new int[3];
```
Declaration   **Array Creation**

num □ → num**[0]** 0

num**[1]** 0

num**[2]** 0

Allocation
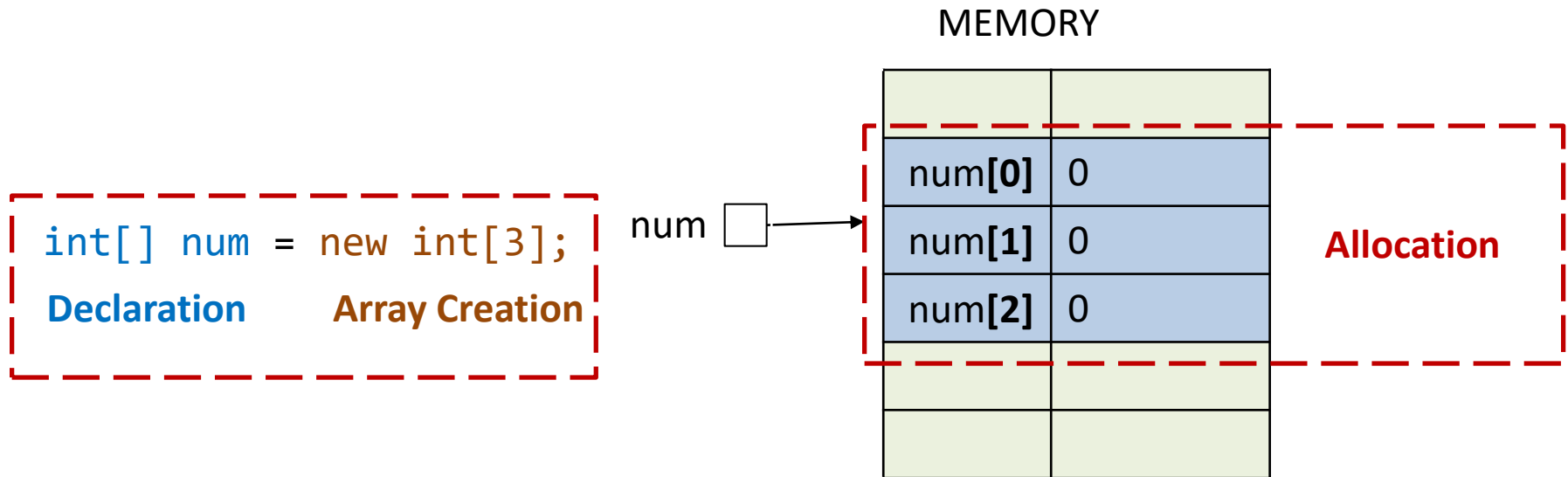
- Array = **fixed-length** data structure storing values of the same type
- Array elements are **auto initialized** with the type's default value:
  - 0 for the numeric primitive-type elements, false for boolean elements and null for references

Array stores values of the same type

```
int[] number = new int[100];        // stores 100 integers


double[] salesTax = new double[10]; // stores 10 doubles


char[] alphabet = new char[26];     // stores 26 characters


Student[] students = new Student[40];   // stores 40 students
```

- The array **size** determines the number of elements in the array.
- The **size** must be specified in the array declaration and it cannot change once the array is created

You may initialize an array explicitly

```
int[] numbers = {1, 2, 3, 4, 5};  // Array initializer
```

MEMORY
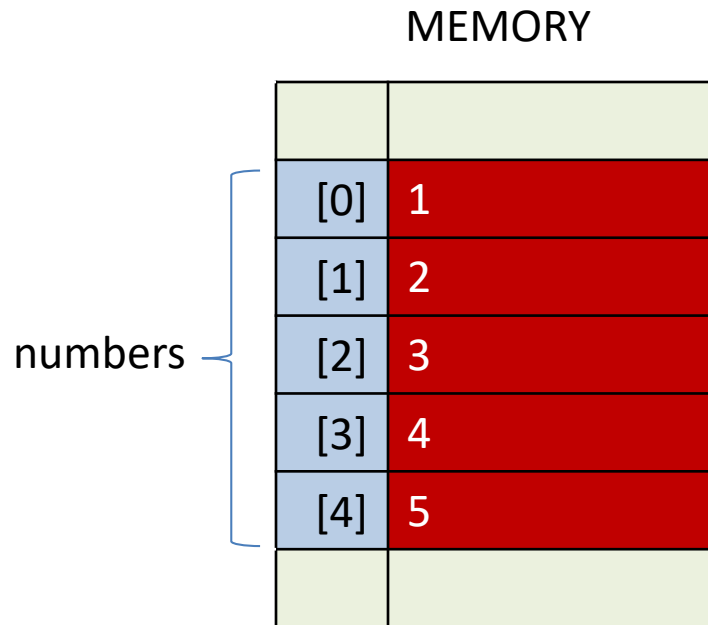
Array elements are indexed

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

MEMORY

[0]	1
[1]	2
[2]	0
[3]	0
[4]	0

numbers

```
numbers[0] = 1;
numbers[1] = 2;
```
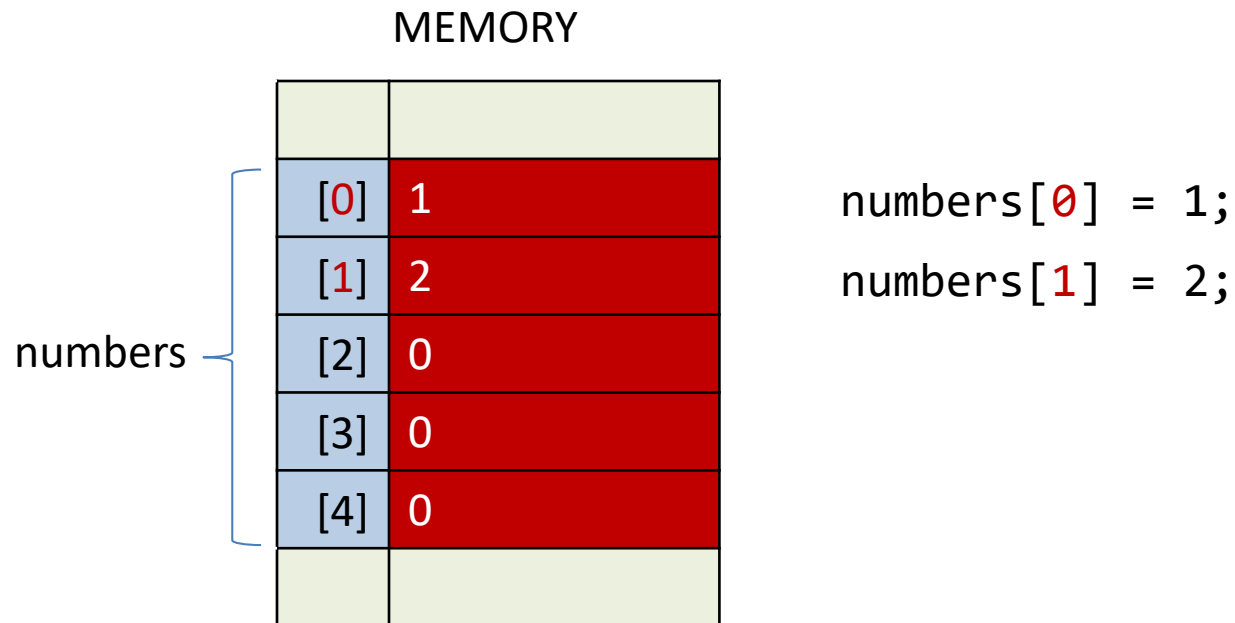
- Array index range is 0 to array size -1

Arrays can be class attributes

```
public class Department {
    private Employee[] employee;
    ...
}
```

Arrays can be local variables

```
 public void getHourlyEmployees() {
     Employee[] hourlyEmployee;
     ...
 }
```

Arrays can be parameters

```
public static void main(String[] args) {
    ...
}
```

Arrays can be return values

```
public Employee[] getEmployees() {
    ...
}
```

Example - Method that returns an array

```java
public int[] initArray(int size, int initValue) {
    int[] array = new int[size];

    for (int i = 0; i < array.length; i++) {
        array[i] = initValue;
    }

    return array;
}
```

Arrays are objects, thus

```
int[] a = {1, 2, 3};
int[] b;

b = a;                    // makes b and a refer to the same
                          // memory location
```

- Arrays are objects so they are **reference types**.
- Array elements can be either primitive or reference types.

Arrays are objects, thus

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};

if (a == b) {...}   // evaluates to false
                    // since a and b refer to two
                    // different memory locations
```

Example - Method that tests for array equality

```java
public boolean areEqual(int[] array1, int[] array2) {
    if (array1.length != array2.length) {
        return false;
    } else {
        for(int i = 0; i < array1.length; i++) {
            if(array1[i] != array2[i])
                return false;
        }// end for
    }// end if
    return true;
}
```

Use linear (sequential) search to locate values

MEMORY

[0]	12
[1]	1
[2]	44
[3]	15
[4]	6

an array

Q: is this the value? A: No

Q: is this the value? A: No

Q: is this the value? A: No

Q: is this the value? A: Yes

Q: is this the value 15 in the array?

Linear Search

```java
    // Returns true if array contains item, false otherwise.
private boolean contains(String[] items, String element) {
        // Using enhanced for loop to iterate through the array
        for(var item : items) {
                if (item.equalsIgnoreCase(element)) {
                        return true;
                }
        }// end for
        return false;
}
```

Enhanced for loop

- The enhanced for loop (also called a "for each" loop) allows you to iterate through the elements of an array or a list without using a counter.

- The syntax of an enhanced for statement is:

```
for {var item : arrayName) {

    statement;

}
```

Multidimensional Arrays

- **Two-dimensional arrays** are often used to represent tables of values with data arranged in *rows* and *columns*.

- Example two-dimensional arrays with 3 rows and 4 columns

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Column index
Row index
Array name

Multidimensional Arrays (Cont.)

- A multidimensional array **b** with 3 rows and 4 columns

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

- A two-dimensional array **b** with 2 rows and 3 columns could be declared and initialized with nested array initializers as follows:

  ```
  int[][] b = {{1, 2, 9}, {3, 4, 8}};
  ```

  - The initial values are *grouped by row* in braces.

  - The number of nested array initializers (represented by sets of braces within the outer braces) determines the number of *rows*.

  - The number of initializer values in the nested array initializer for a row determines the number of *columns* in that row.

Lists

- Problem with arrays
  - You must know the array size when you create the array
  - Array size cannot change once created.

- Solution:
  - Use **ArrayList**: they stretch as you add elements to them or shrink as you remove elements from them
  - Similar to arrays + allow **Dynamic resizing**

ArrayList methods

- Create empty list
  `new ArrayList<>()`
- Add entry to end
  `add(value)`
- Retrieve n^{th} element
  `get(index)`
- Check if element exists in list
  `contains(element)`
- Remove element
  `remove(index)` or `remove(element)`
- Get the number of elements
  `size()`
- Remove all elements
  `clear()`

ArrayList Example

```java
import java.util.*;  // Don't forget this import

public class ListTest2 {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

This tells Java that the list will contain only strings.

Variable-Length Argument Lists

- Variable-length argument lists can be used to create methods that receive an **unspecified** number of arguments.

  - Parameter type followed by an **ellipsis (...)** indicates that the method receives a variable number of arguments of that particular type.

  - The ellipsis can occur only once at the end of a parameter list.

- A variable-length argument list **is treated as an array** within the method body. The number of arguments in the array can be obtained using the array's length attribute.

Variable-Length Argument Lists - Example

```java
// Variable-Length Argument Lists - Example
public static double average(double... numbers) {
    double total = 0.0;
    for(var num : numbers) {
        total += num;
    }
    return total / numbers.length;
}

public static void main(String[] args) {
    double avg = average(4, 6, 2);
    System.out.println(avg);
}
```