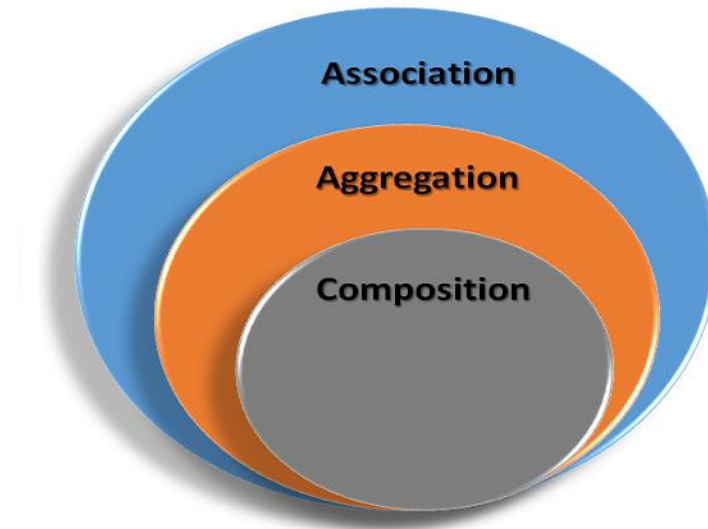




## Relations between Classes

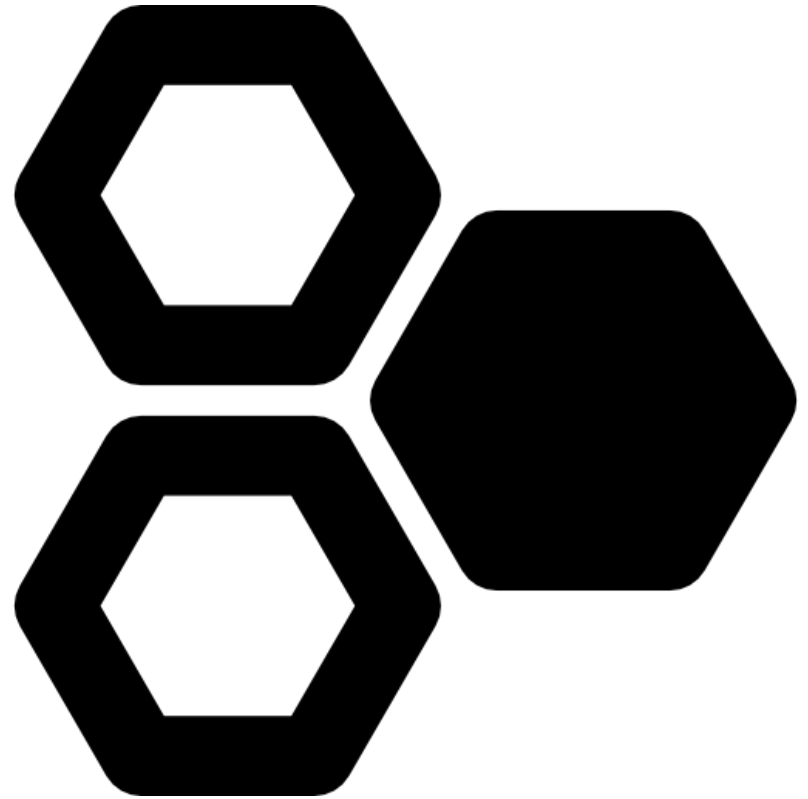


**Dr. Abdelkarim Erradi**  
**CSE@QU**

# Outline

- **Relations between Classes**
- **Introduction to Arrays and Lists**
- **Java Packages**
- **Enumeration**
- **Exceptions**

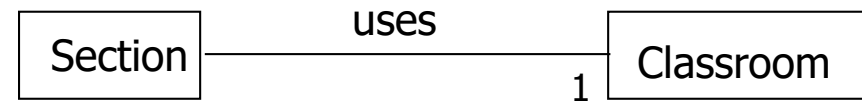
# Relations between Classes



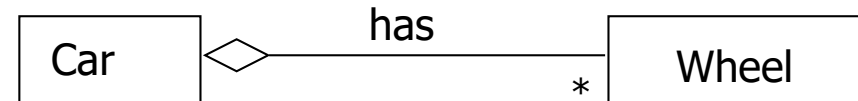
# Relations between Classes

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

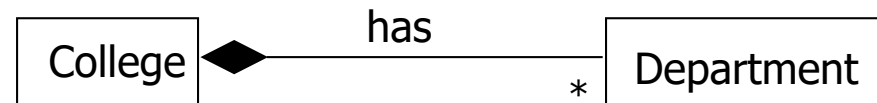
- **Association** (uses without ownership)



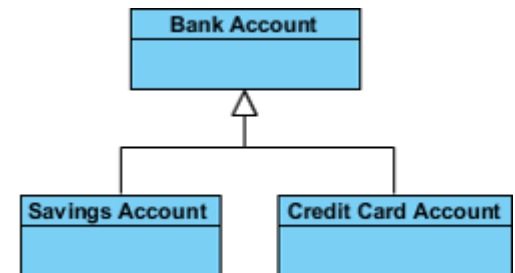
- **Aggregation** (has-a + Whole-Part relationship)



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



- **Inheritance** (is-a relation)



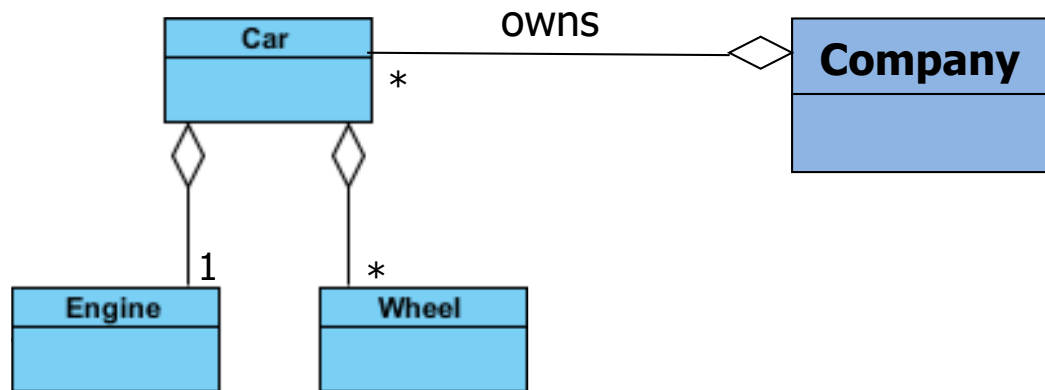
# Association

- 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

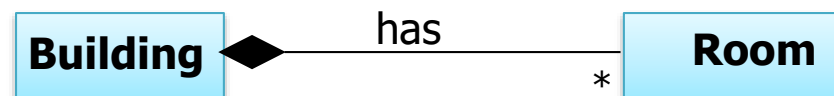
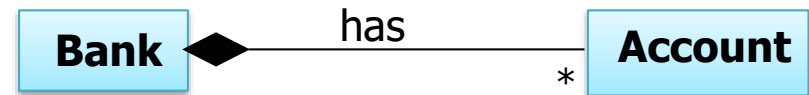
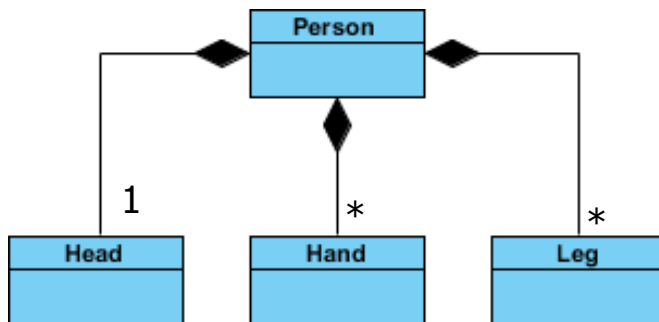


# Aggregation vs. Composition

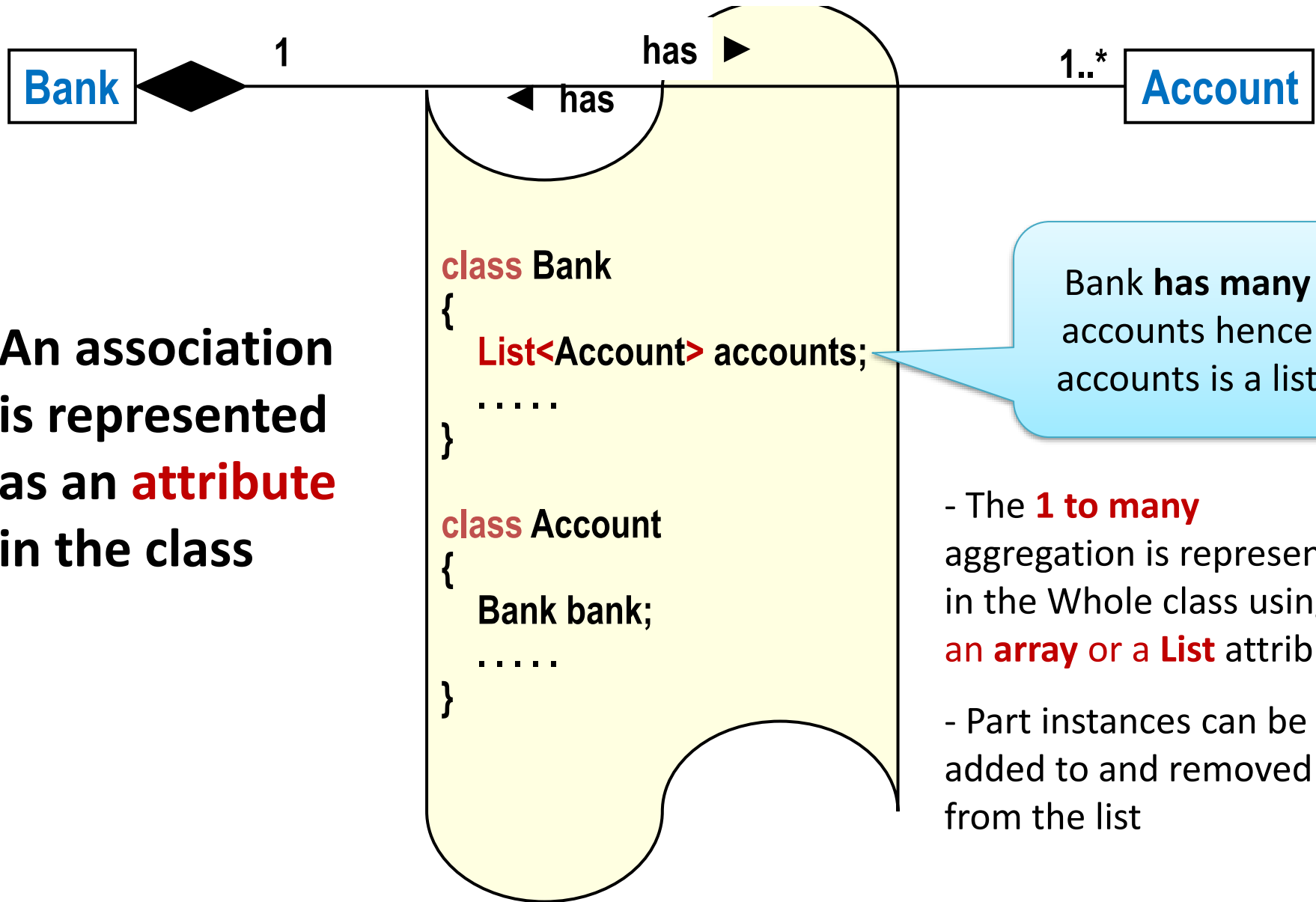
- **Aggregation** = WHOLE-PART relationship. PART can exist without the WHOLE.



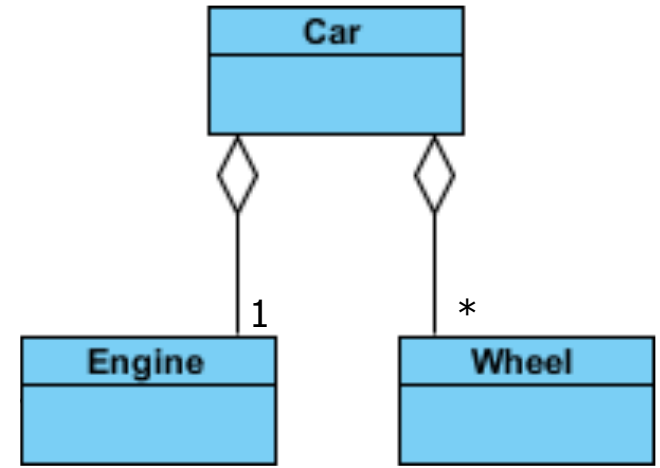
- **Composition** = WHOLE-PART relationship. PART cannot meaningfully exist without the WHOLE



# Implementation of bidirectional association



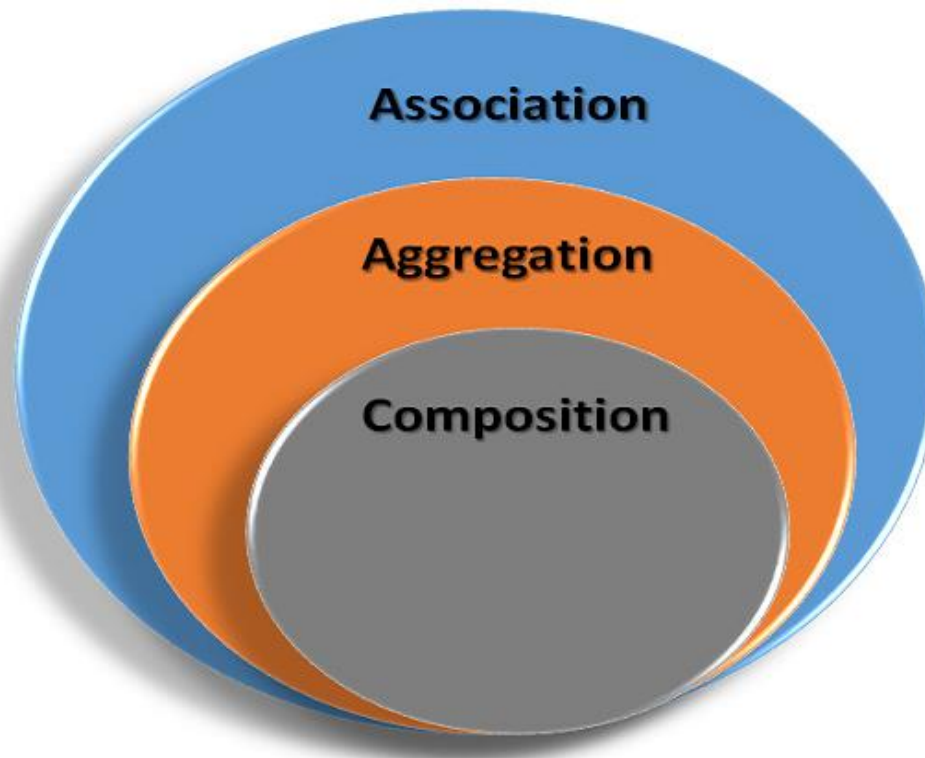
```
public class Car {  
    private Engine engine;  
    private List<Wheel> wheels;  
  
    public Car(Engine engine){  
        this.engine = engine;  
        this.wheels = new ArrayList<>();  
    }  
  
    public addWheel(Wheel wheel){  
        wheels.add(wheel);  
    }  
}  
  
class Engine {  
    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

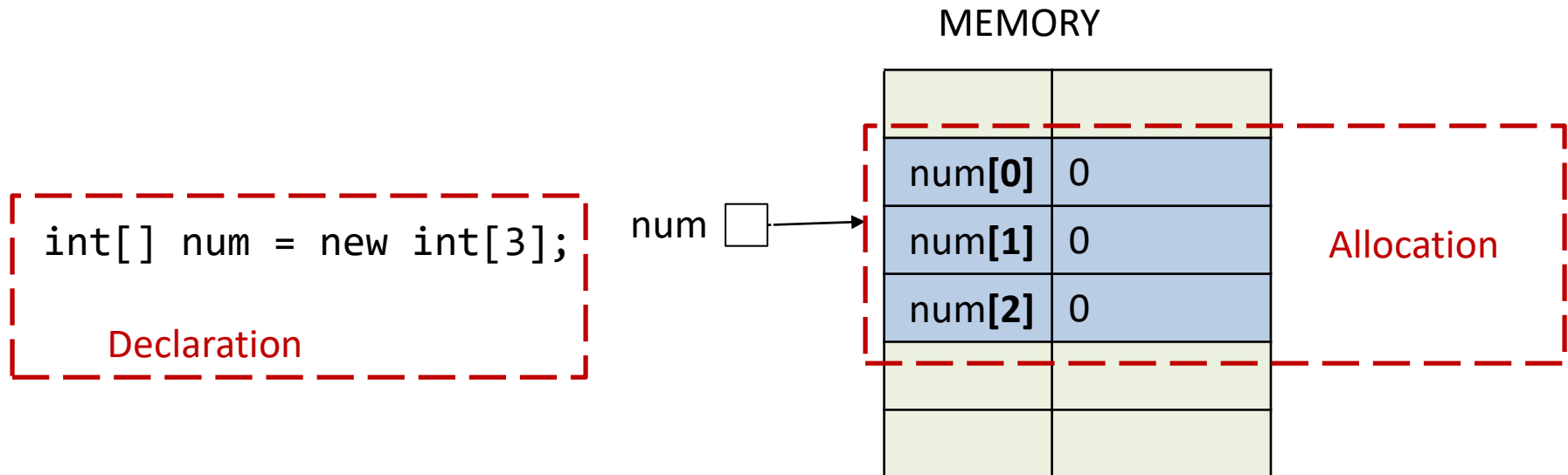
```
int num1 = 10;
```

```
int num2 = 20;
```

```
int num3 = 30;
```

|             |    |
|-------------|----|
|             |    |
| <b>num1</b> | 10 |
| <b>num2</b> | 20 |
| <b>num3</b> | 30 |
|             |    |
|             |    |

# An array object stores multiple values



# Array objects can hold any type of object

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

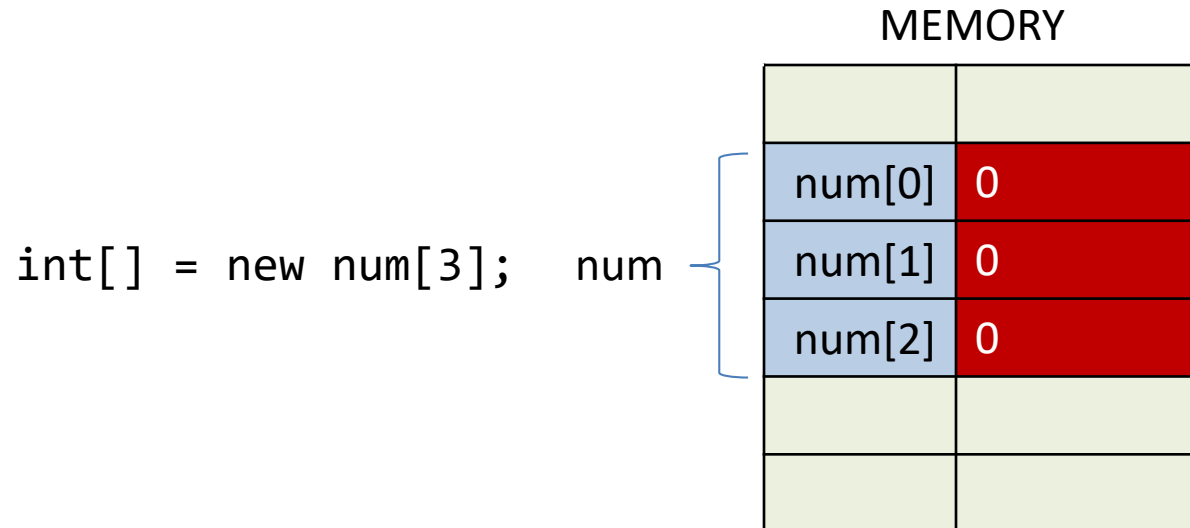
```
double[] salesTax = new double[10];    // stores 10 doubles
```

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

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

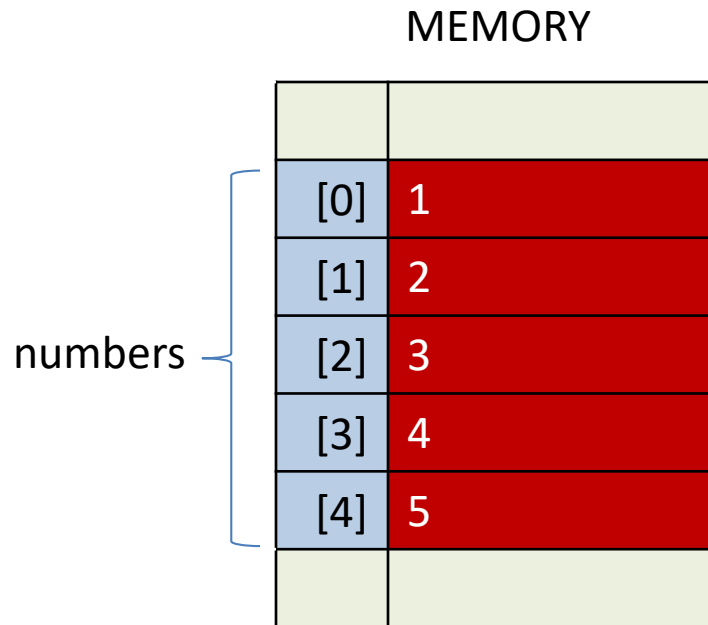
The **size** of the array determines the number of elements in the array.

# Array elements are initialized to the type's default value



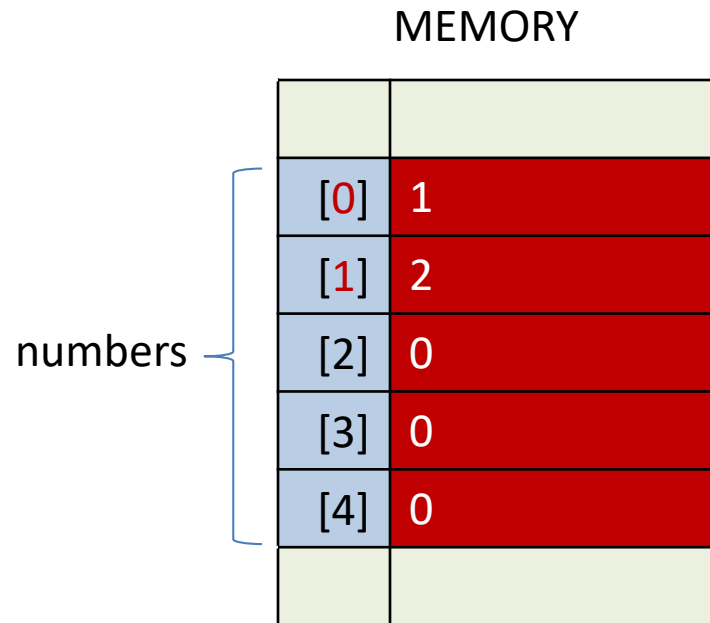
# You may initialize an array explicitly

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



# Array elements are indexed

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



numbers[0] = 1;

numbers[1] = 2;



# Arrays can be instance variables

```
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

```
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
```

# Example method that copies an array

```
public void copyArray(int[] source, int[] target) {  
    // Both arrays must be the same size.  
    target = new int[source.length];  
    for (int i = 0; i < source.length; i++) {  
        target[i] = source[i];  
    }  
}
```

# 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

```
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

```
// Returns true if array contains item, false otherwise.
private boolean contains(String[] items, String item) {
    for(int i = 0; i < items.length; i++) {
        if (items[i].equalsIgnoreCase(item)) {
            return true;
        }
    } // end for
    return false;
}
```



# Lists

- Problem
  - You must know the array size when you create the array
    - Although Java arrays are better than C++ arrays since the size does not need to be a compile-time constant
  - Array size cannot change once created.
- Solution:
  - Use **ArrayList**: they stretch as you add elements to them

# ArrayList methods

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

# ArrayList Example

```
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 your list will contain only strings.



# Packages



# Packages

- Packages in Java are a **way of grouping related classes** together:
  - Classes performing a specific set of tasks or providing similar functionality.
- **Package = directory.** A package name is the same as the directory (folder) name which contains the .java files.
- Two main reasons packages are used:
  - Code organization: grouping functionally related classes into a package to make it easier to find and use classes
  - Avoid names collision: distinguish between classes with the same name but belong to different packages

# Built-in Packages



- Java fundamental classes are in *java.lang*, classes for reading and writing (input and output) are in *java.io*, lists and collections in *java.util* and so on.
- To use a class from a package, first **import** it. E.g.,

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

# Creating a Package

- To create a package, you add **package** **statement** with the package name at the top of every source file that you want to include in the package

```
package quBank;
```

```
public class Account {  
    // OOP Principle of Encapsulation:  
    all attributes are private  
    private int id;  
    private String name;  
    private String type;  
    private double balance;  
  
    ...  
}
```

- All .java files in **quBank** package will be saved in quBank folder
- Package names are usually written in lowercase



# Enumeration

```
enum LightState {...}
```



# Enumerations

- The basic enum type defines a set of constants represented as unique identifiers
- An enum type is declared with an **enum declaration**, which is a comma-separated list of enum constants
- The declaration may optionally include constructors, attributes and methods

# Enumerations (Cont.)

- Each `enum` declaration declares an `enum` class with the following restrictions:
  - `enum` constants are implicitly `final`, because they declare constants that shouldn't be modified.
  - `enum` constants are implicitly `static`.
  - Any attempt to create an object of an `enum` type with operator `new` results in a compilation error.
  - `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and the condition of an `if` statement.
- For every `enum`, the compiler generates the `static` method `values` that returns an array of the `enum`'s constants.
- When an `enum` constant is converted to a `String`, the constant's identifier is used as the `String` representation.

# enum is actually a class



## <EnumDemo.java>

```
public class EnumDemo {
    enum LightState {
        // Each object is initialized to a color.
        OFF("black"),
        ON("white"),
        DIMMED("gray"),
        FLICKERING("red");

        private final String colorField;

        // Private constructor to set the color.
        private LightState(String color) {
            colorField = color;
        }

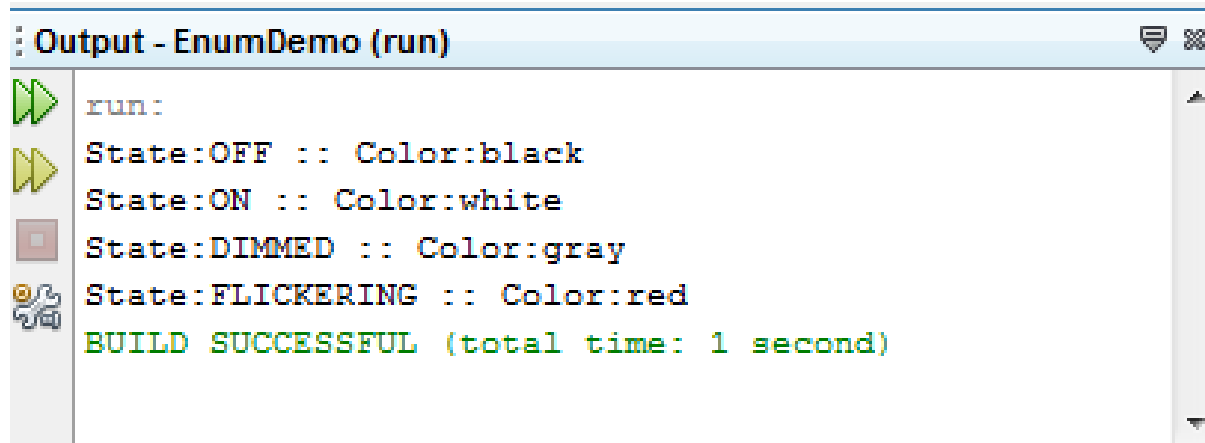
        // Public accessor to get color.
        public String getColor() {
            return colorField;
        }
    }

    public static void main(String[] args) {
        LightState off = LightState.OFF;
        LightState on = LightState.ON;
        LightState dimmed = LightState.DIMMED;
        LightState flickering = LightState.FLICKERING;
    }
}
```

You can enhance the enum class with instance variables and methods

# <EnumDemo.java>

```
System.out.println("State:" + off.toString() +  
                    " :: Color:" + off.getColor());  
System.out.println("State:" + on.toString() +  
                    " :: Color:" + on.getColor());  
System.out.println("State:" + dimmed.toString() +  
                    " :: Color:" + dimmed.getColor());  
System.out.println("State:" + flickering.toString() +  
                    " :: Color:" + flickering.getColor());  
    }// end main()  
}// end EnumDemo
```



The screenshot shows an IDE's Output window titled "Output - EnumDemo (run)". The window contains the following text:

```
run:  
State:OFF :: Color:black  
State:ON :: Color:white  
State:DIMMED :: Color:gray  
State:FLICKERING :: Color:red  
BUILD SUCCESSFUL (total time: 1 second)
```

On the left side of the output window, there are four icons: a green play button, a yellow play button, a red square, and a blue icon with a magnifying glass.



Exceptions

**Error.**

# Throwing Exceptions

```
1 // Time1.java
2 // Time1 class declaration maintains the time in 24-hour format.
3
4 public class Time1 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // set a new time value using universal time; throw an
10    // exception if the hour, minute or second is invalid
11    public void setTime(int hour, int minute, int second) {
12        // validate hour, minute and second
13        if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
14            second < 0 || second >= 60) {
15            throw new IllegalArgumentException(
16                "hour, minute and/or second was out of range");
17        }
18
19        this.hour = hour;
20        this.minute = minute;
21        this.second = second;
22    }
```

# Throwing Exceptions

- Method `setTime` declares three `int` parameters and uses them to set the time.
- Lines 13–14 test each argument to determine whether the value is outside the proper range.
- For incorrect values, `setTime` throws an exception of type `IllegalArgumentException`
  - Notifies the client code that an invalid argument was passed to the method.
  - The **throw statement** creates a new object of type `IllegalArgumentException` and specifies a custom error message.
  - `throw` statement immediately terminates method `setTime` and the exception is returned to the calling method that attempted to set the time.



# try and catch

---

```
18 // attempt to set time with invalid values
19 try {
20     time.setTime(99, 99, 99); // all values out of range
21 }
22 catch (IllegalArgumentException e) {
23     System.out.printf("Exception: %s%n%n", e.getMessage());
24 }
25
26 // display time after attempt to set invalid values
27 displayTime("After calling setTime with invalid values", time);
28 }
29
30 // displays a Time1 object in 24-hour and 12-hour formats
31 private static void displayTime(String header, Time1 t) {
32     System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33         header, t.toUniversalString(), t.toString());
34 }
35 }
```

---

Lines 19 to 24 use **try...catch** to catch and handle the exception (e.g., display the error message to the user)

# Banking System Example

QuBank 🔍

| BankUI                                |
|---------------------------------------|
| <u>+main(args : String []) : void</u> |

**This is the main  
class to run the App**

| Account   |
|---|
| <<Property>> -accountNo : int                                     |
| <<Property>> -accountName : String                                |
| <<Property>> -balance : double                                    |
| +Account(accountNo : int, accountName : String, balance : double) |
| +Account(accountNo : int, accountName : String)                   |
| +deposit(amount : double) : String                                |
| +withdraw(amount : double) : String                               |

Bank has  
many  
Accounts

| Bank  |
|---|
| <u>-lastAccountNo : int = 0</u>                             |
| <u>-accounts : Account = new ArrayList&lt;&gt;()</u>        |
| <u>+addTestAccounts() : void</u>                            |
| <u>+addAccount(account : Account) : void</u>                |
| <u>+getAccount(accountNo : int) : Account</u>               |
| <u>+getBalance(accountNo : int) : double</u>                |
| <u>+deposit(accountNo : int, amount : double) : String</u>  |
| <u>+withdraw(accountNo : int, amount : double) : String</u> |
| <u>+getFormattedBalance(accountNo : int) : String</u>       |



# Bookstore System example

QuBookstore 🔍

| BookStoreUI   |
|---|
| <pre>+main(args : String []) : void +addItemsToShoppingCart() : ShoppingCart +displayShoppingCart(shoppingCart : ShoppingCart) : void</pre> |

**This is the main class to run the App**

| DataEntryUtils   |
|--|
| <pre>+getString(scanner : Scanner, prompt : String) : String +getInt(scanner : Scanner, prompt : String) : int +getInt(scanner : Scanner, prompt : String, min : int, max : int) : int +getDouble(scanner : Scanner, prompt : String) : double +getDouble(scanner : Scanner, prompt : String, min : double, max : double) : double</pre> |

**This is a utility class to ease data entry**

**A CartItem is for one particular book**

| CartItem  |
|---|
| <pre>&lt;&lt;Property&gt;&gt; -quantity : int &lt;&lt;Property&gt;&gt; -book : Book +CartItem() +CartItem(book : Book, quantity : int) +getTotal() : double +getFormattedTotal() : String</pre> |

| ShoppingCart  |
|---|
| <pre>&lt;&lt;Property&gt;&gt; -cartItems : CartItem +ShoppingCart() +addItem(cartItem : CartItem) : void +getTotal() : double +getFormattedTotal() : String</pre> |

**A Shopping Cart contains 1 or many items**

| Book  |
|---|
| <pre>&lt;&lt;Property&gt;&gt; -code : String &lt;&lt;Property&gt;&gt; -description : String &lt;&lt;Property&gt;&gt; -price : double +Book() +Book(code : String, description : String, price : double) +getFormattedPrice() : String</pre> |

| BookCatalog  |
|--|
| <pre>-books : Book = new ArrayList&lt;&gt;() +addTestBooks() : void +getBook(bookCode : String) : Book</pre> |

**The BookCatlog contains many books**