13. Arrays and Collections

5 Nov 2015

Objectives

- Manipulate a collection of data values, using an array.
- Declare and use an array of primitive data types.
- Declare and use an array of objects.
- Define a method that accepts an array as its parameter and a method that returns an array.

Objectives

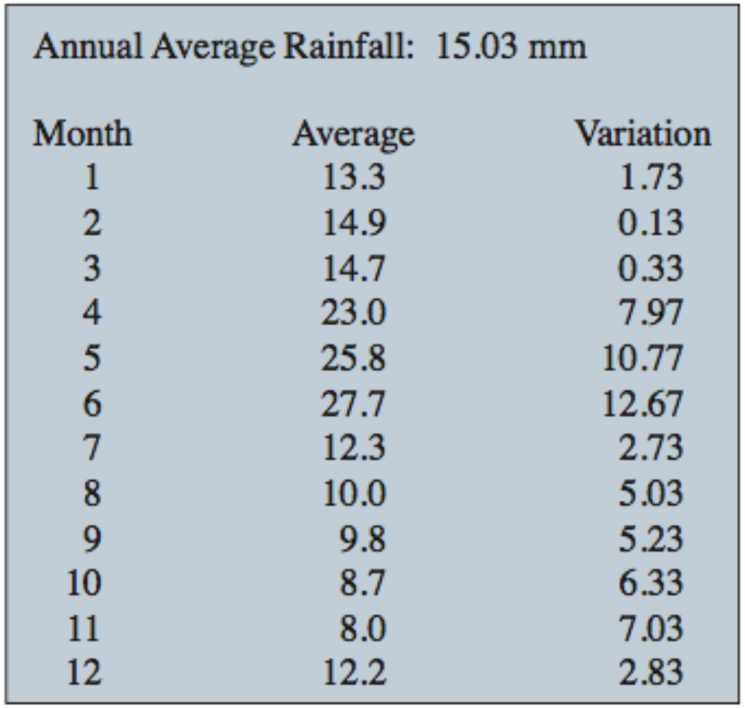
- Describe how a two-dimensional array is implemented as an array of arrays.
- Manipulate a collection of objects, using lists and maps.

Array Basics

- An array is a collection of data values of the same type. For example, we may declare an array consisting of double, but not an array consisting of both int and double.

Array Basics

- Now suppose we want to compute the difference between the annual and monthly averages for every month



Annual Average Rainfall: 15.03 mm

Month	Average	Variation
1	13.3	1.73
2	14.9	0.13
3	14.7	0.33
4	23.0	7.97
5	25.8	10.77
6	27.7	12.67
7	12.3	2.73
8	10.0	5.03
9	9.8	5.23
10	8.7	6.33
11	8.0	7.03
12	12.2	2.83

Array Basics

- To compute the difference between the annual and monthly averages, we need to remember the 12 monthly rainfall averages. Instead of using 12 variables januaryRainfall, februaryRainfall, and so forth to solve this problem, we use an array.

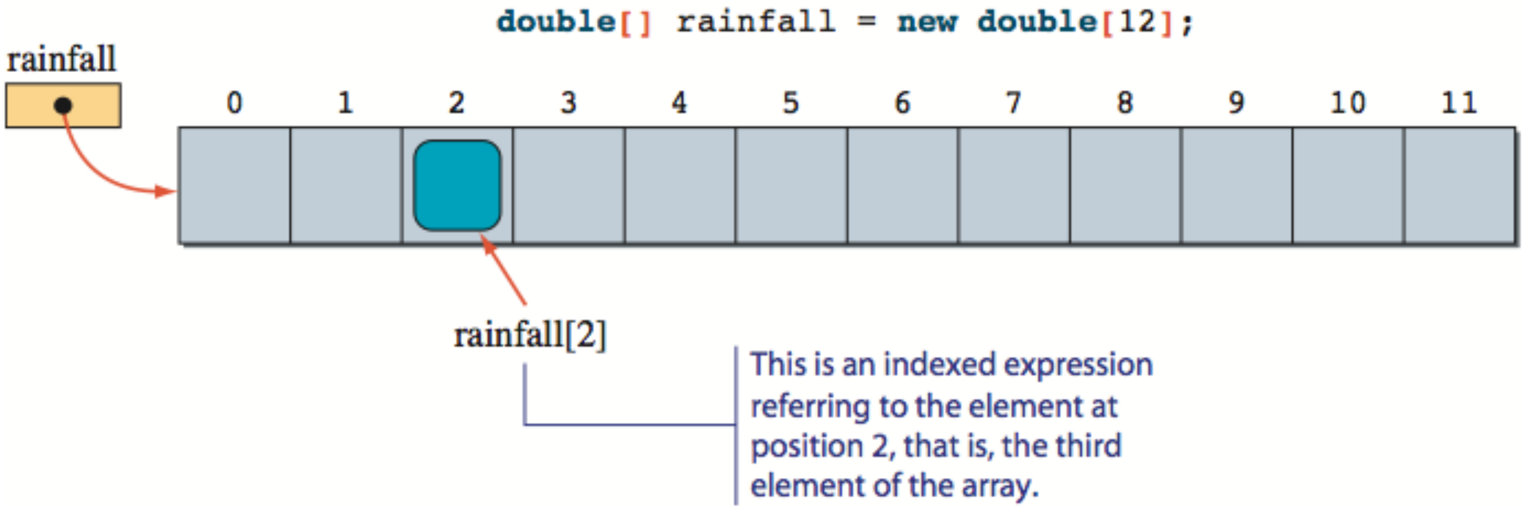
Array Basics

- In Java, an array is a reference data type. Unlike the primitive data type, the amount of memory allocated to store an array varies, depending on the number and type of values in the array. We use the “new” operator to allocate the memory to store the values in an array

Array Basics

- The following declares an array of double with size 12:

```
double[] rainfall = new double[12];

double rainfall[] = new double[12];
```



This is an indexed expression referring to the element at position 2, that is, the third element of the array.

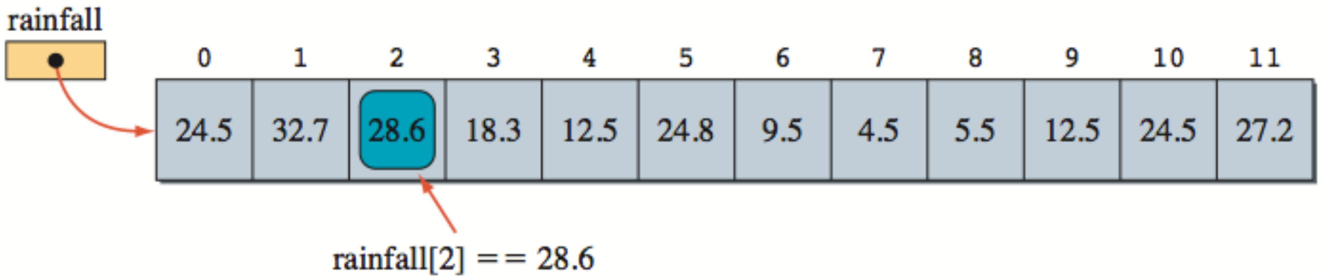
Array Basics

- The number 12 designates the size of the array-the number of values the array contains.
- We use a single identifier to refer to the whole collection and use an “indexed expression” to refer to the individual values of the collection.

Array Basics

- Zero-based indexing is used to indicate the positions of an element in the array. They are numbered 0, 1, 2, ..., and size – 1
- For example, to refer to the third element of the rainfall array

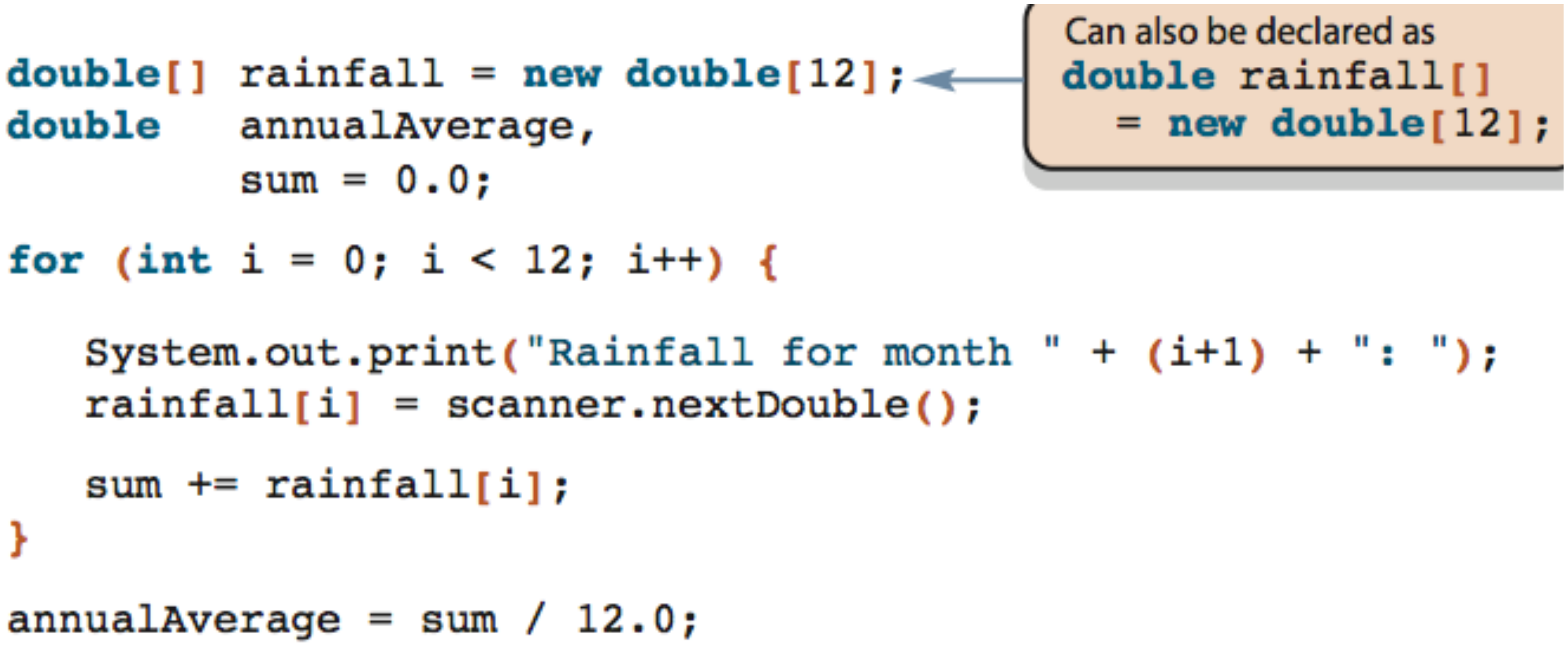
`rainfall[2]`



0	1	2	3	4	5	6	7	8	9	10	11
24.5	32.7	28.6	18.3	12.5	24.8	9.5	4.5	5.5	12.5	24.5	27.2

Array Basics

- Using the rainfall array, we can input 12 monthly averages and compute the annual average as



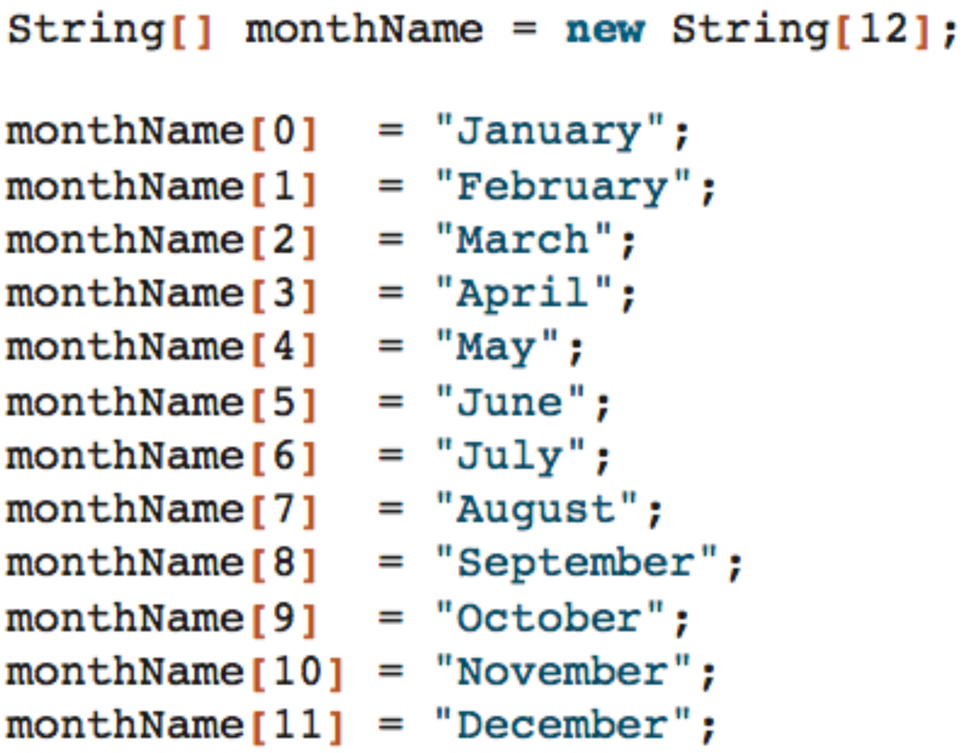
```
double[] rainfall = new double[12];
double   annualAverage,
         sum = 0.0;

for (int i = 0; i < 12; i++) {

   System.out.print("Rainfall for month " + (i+1) + ": ");
   rainfall[i] = scanner.nextDouble();

   sum += rainfall[i];
}

annualAverage = sum / 12.0;
```

Can also be declared as

```
double rainfall[]
   = new double[12];
```

Array Basics

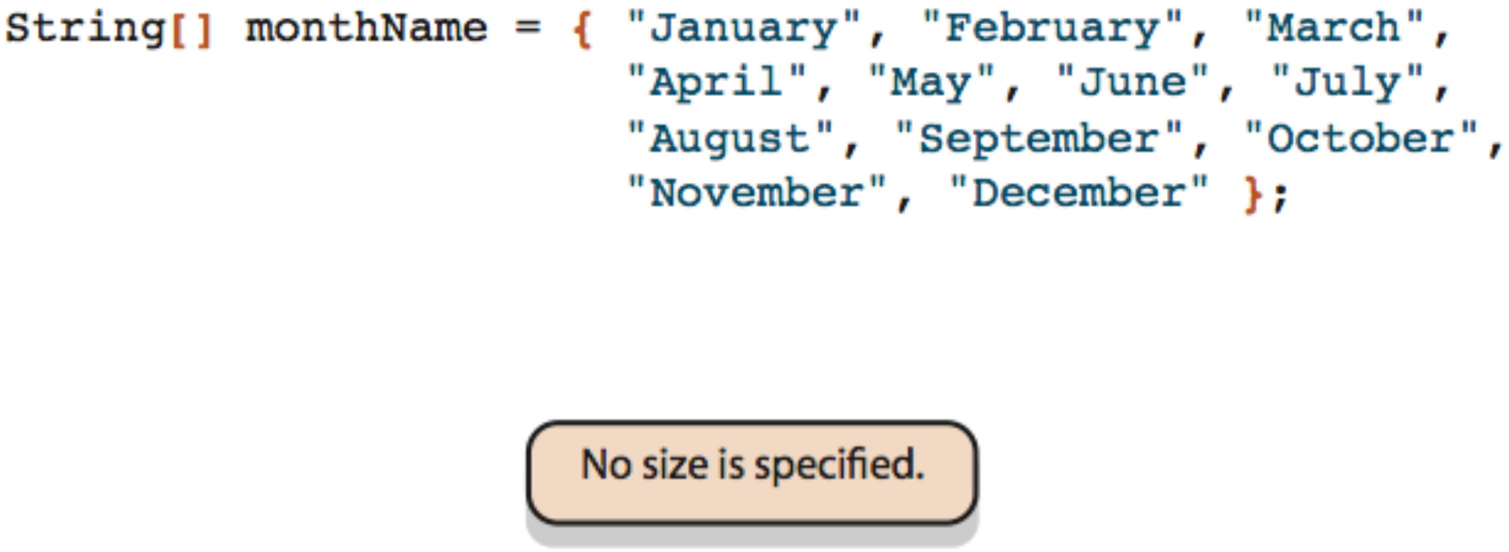
- We can assigning array elements 2 ways
    1. assigning array elements individually



```
String[] monthName = new String[12];

monthName[0]  = "January";
monthName[1]  = "February";
monthName[2]  = "March";
monthName[3]  = "April";
monthName[4]  = "May";
monthName[5]  = "June";
monthName[6]  = "July";
monthName[7]  = "August";
monthName[8]  = "September";
monthName[9]  = "October";
monthName[10] = "November";
monthName[11] = "December";
```

Array Basics

2. initialize the array at the time of declaration



```
String[] monthName = { "January", "February", "March",
                       "April", "May", "June", "July",
                       "August", "September", "October",
                       "November", "December" };
```

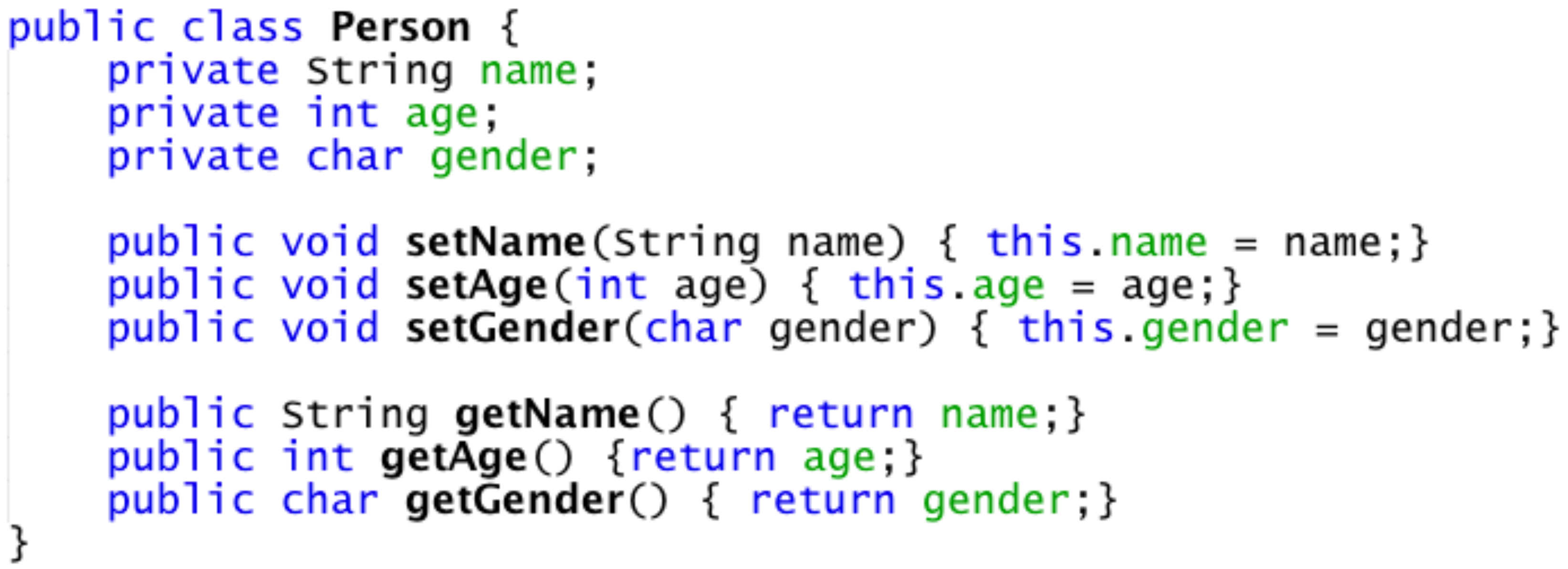
No size is specified.

Arrays of Objects

- Array elements are not limited to primitive data types. Indeed, since a String is actually an object, we have already seen an example of an array of objects. In this section we will explore arrays of objects in detail.

Arrays of Objects

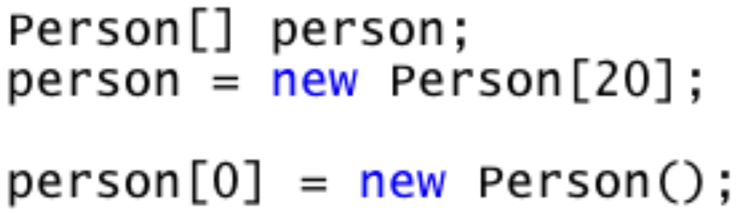
- To illustrate the processing of an array of objects, we will use the Person class in the following examples.



```
public class Person {
    private String name;
    private int age;
    private char gender;

    public void setName(String name) { this.name = name;}
    public void setAge(int age) { this.age = age;}
    public void setGender(char gender) { this.gender = gender;}

    public String getName() { return name;}
    public int getAge() {return age;}
    public char getGender() { return gender;}
}
```

Arrays of Objects

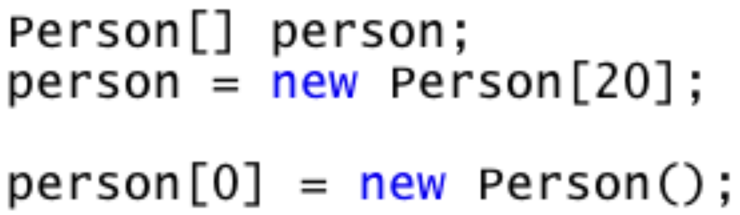
- Now let’s study how we can create and manipulate an array of Person objects. The following are a declaration and a creation of an array of Person objects.



```
Person[] person;
person = new Person[20];

person[0] = new Person();
```

Arrays of Objects

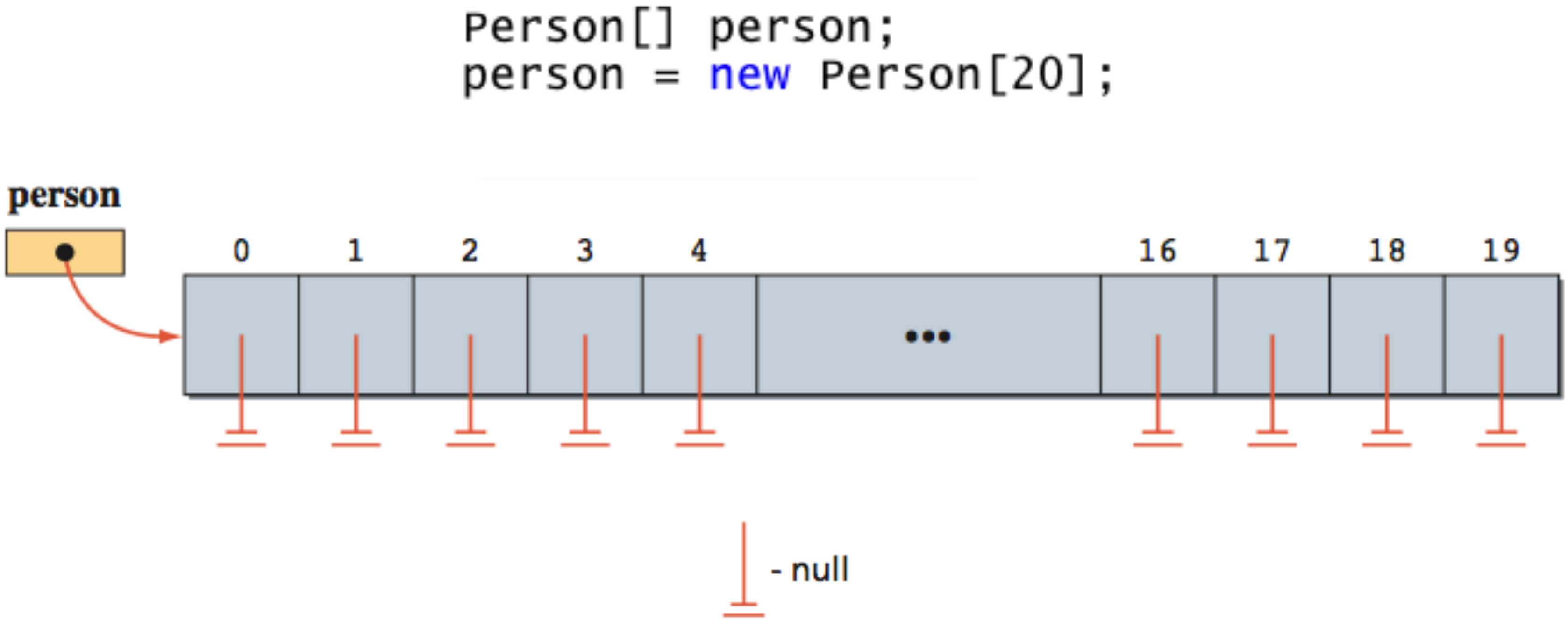
- Now let’s study how we can create and manipulate an array of Person objects. The following are a declaration and a creation of an array of Person objects.



```
Person[] person;
person = new Person[20];

person[0] = new Person();
```

Arrays of Objects

– An array of Person objects after the array is created.

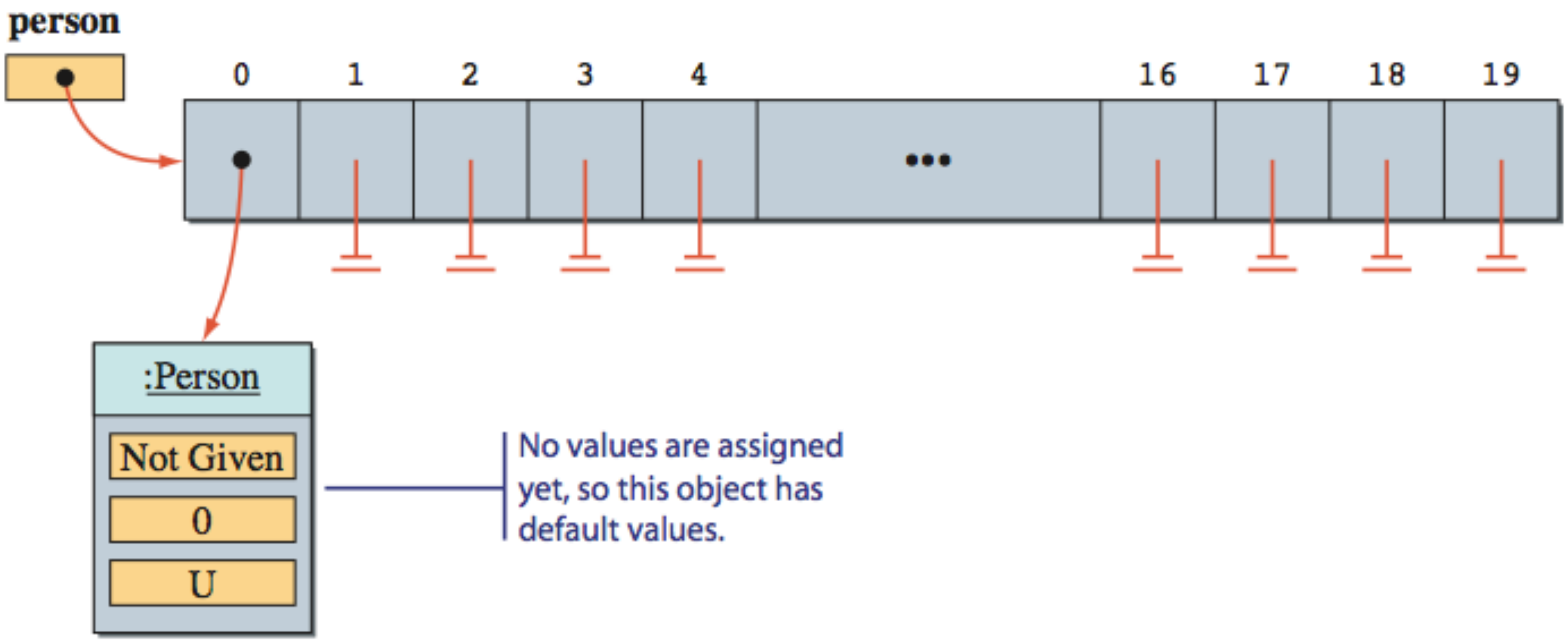


```
Person[] person;
person = new Person[20];
```

Arrays of Objects

– The person array with one Person object added to it.

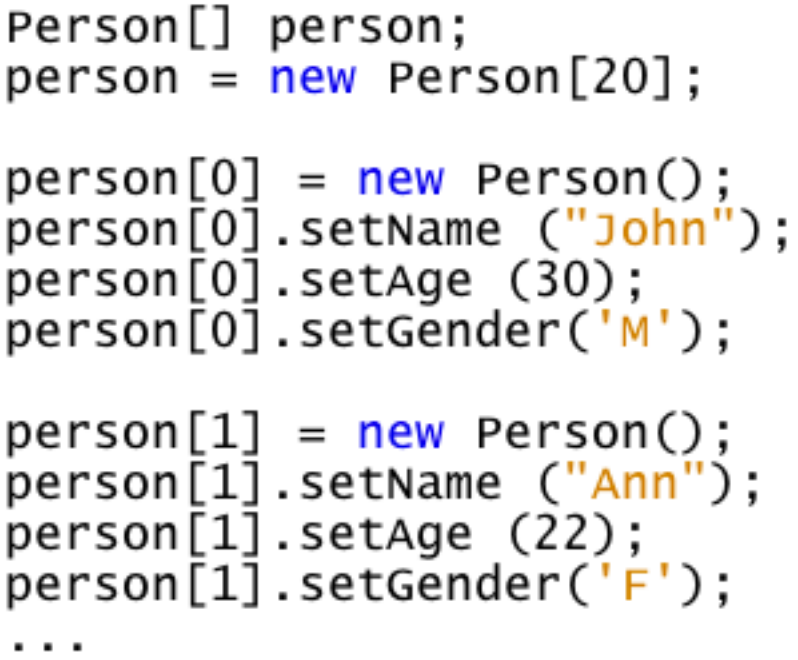
```
person[0] = new Person();
```



No values are assigned yet, so this object has default values.

Arrays of Objects

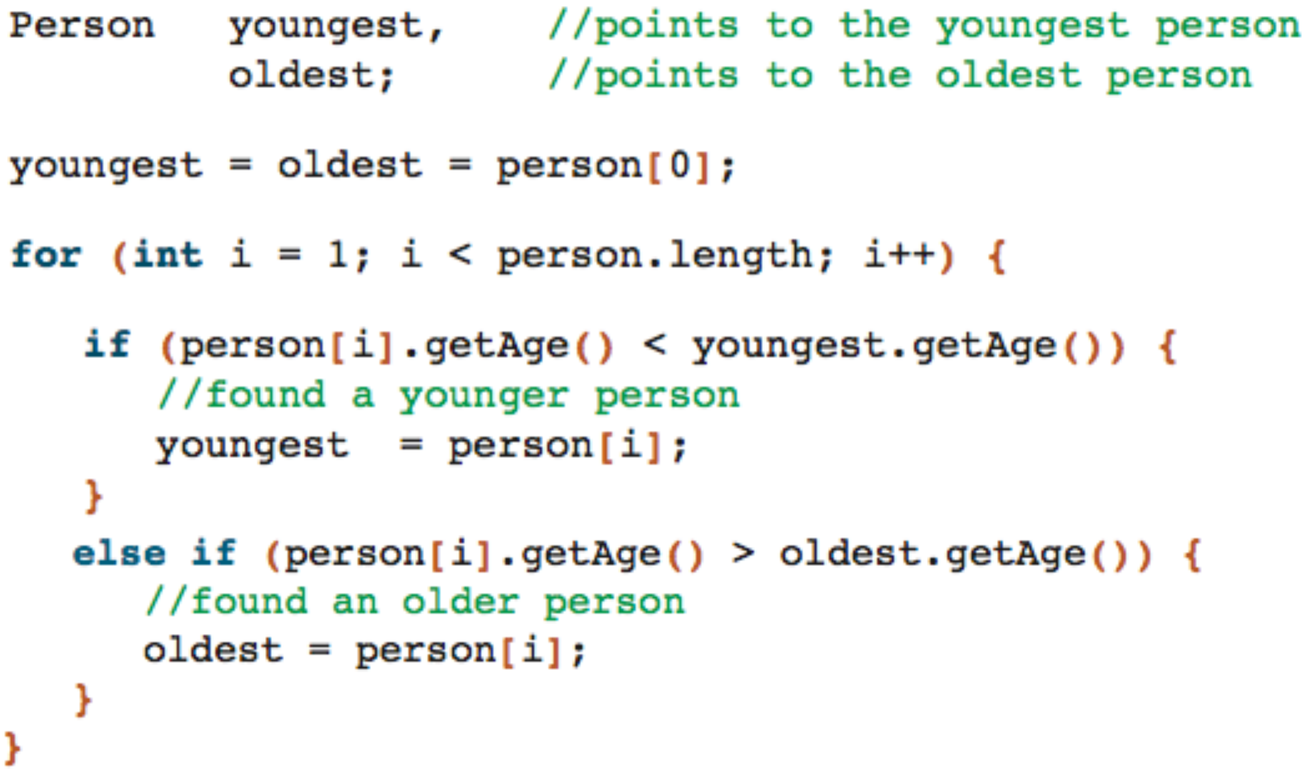
- To focus on array processing, we assume that the person array is already declared and created.



```
Person[] person;
person = new Person[20];

person[0] = new Person();
person[0].setName ("John");
person[0].setAge (30);
person[0].setGender('M');

person[1] = new Person();
person[1].setName ("Ann");
person[1].setAge (22);
person[1].setGender('F');
...
```

Arrays of Objects

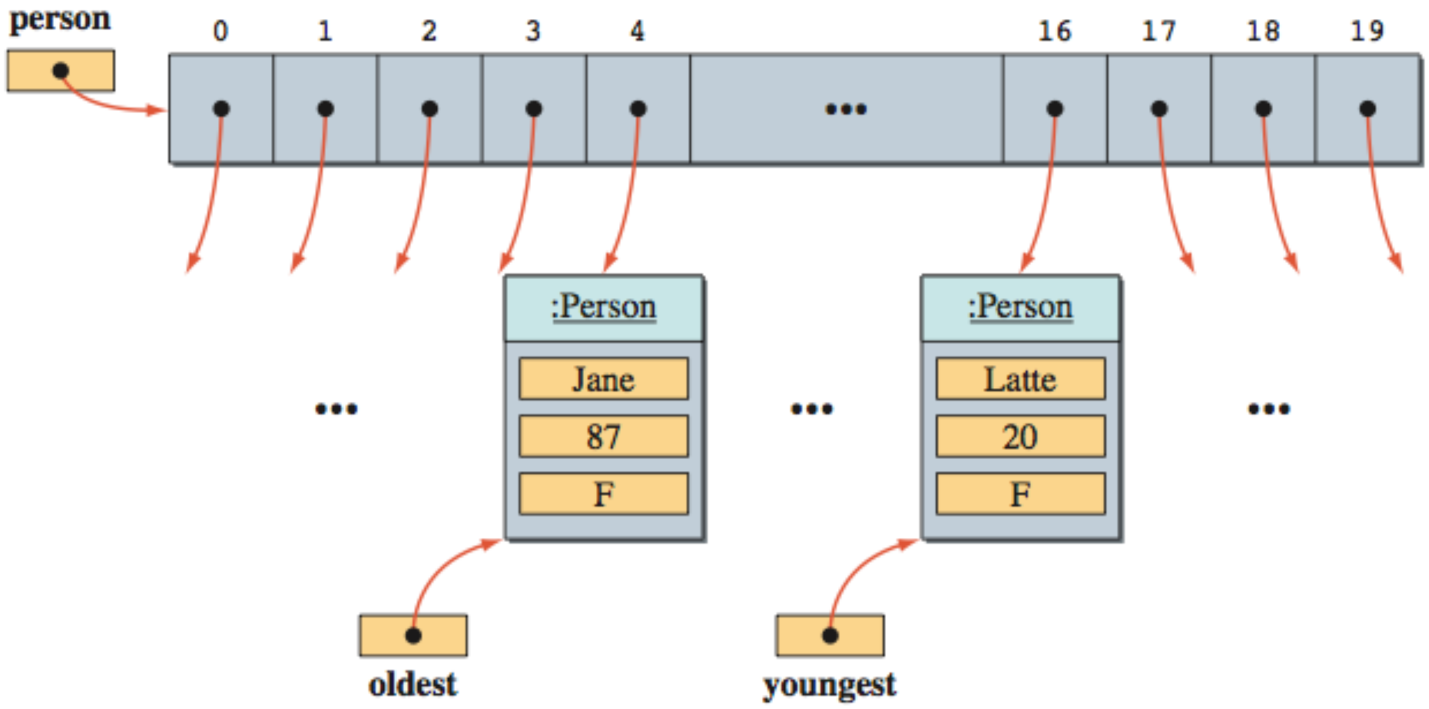
- To find the youngest and the oldest persons, we can execute



```
Person    youngest,     //points to the youngest person
          oldest;       //points to the oldest person

youngest = oldest = person[0];

for (int i = 1; i < person.length; i++) {

   if (person[i].getAge() < youngest.getAge()) {
      //found a younger person
      youngest  = person[i];
   }
   else if (person[i].getAge() > oldest.getAge()) {
      //found an older person
      oldest = person[i];
   }
}
```

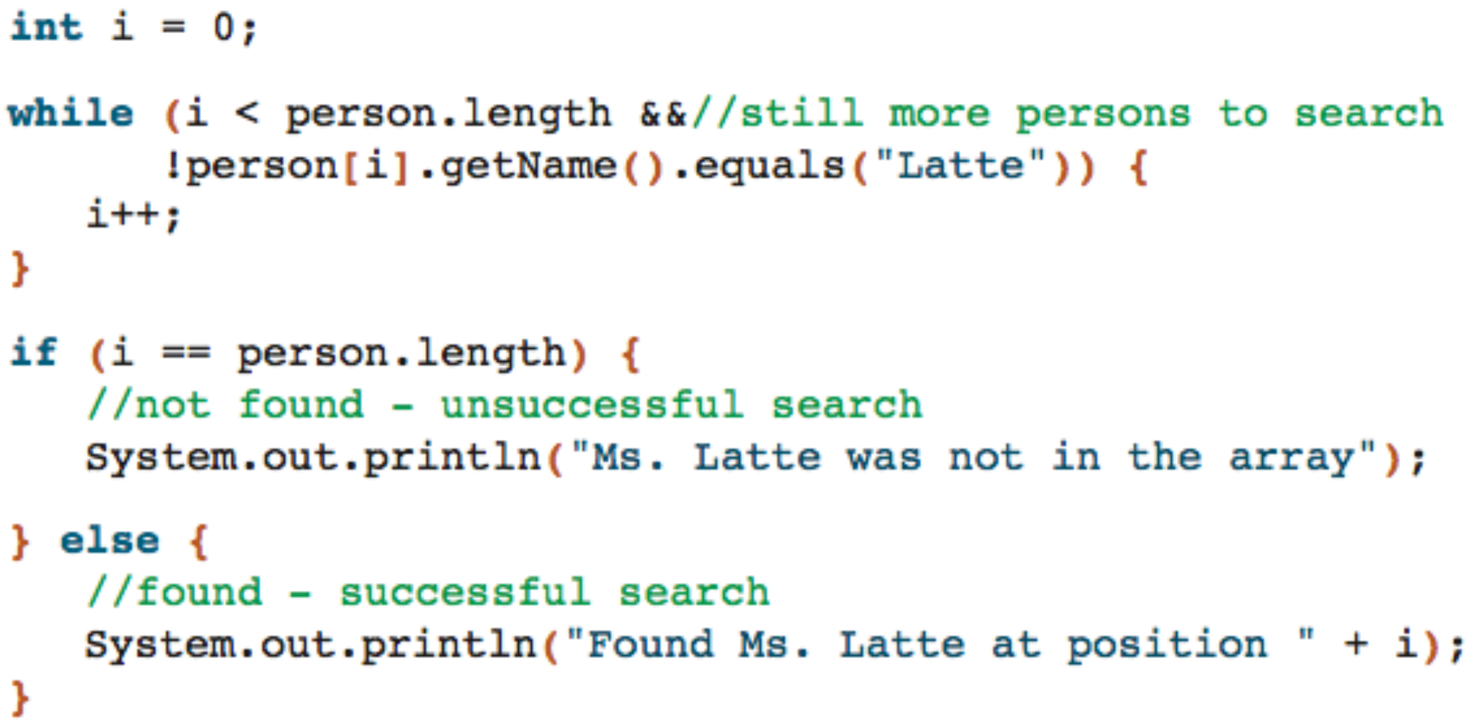
Arrays of Objects

– An array of Person objects with two Person variables (youngest and oldest).



Arrays of Objects

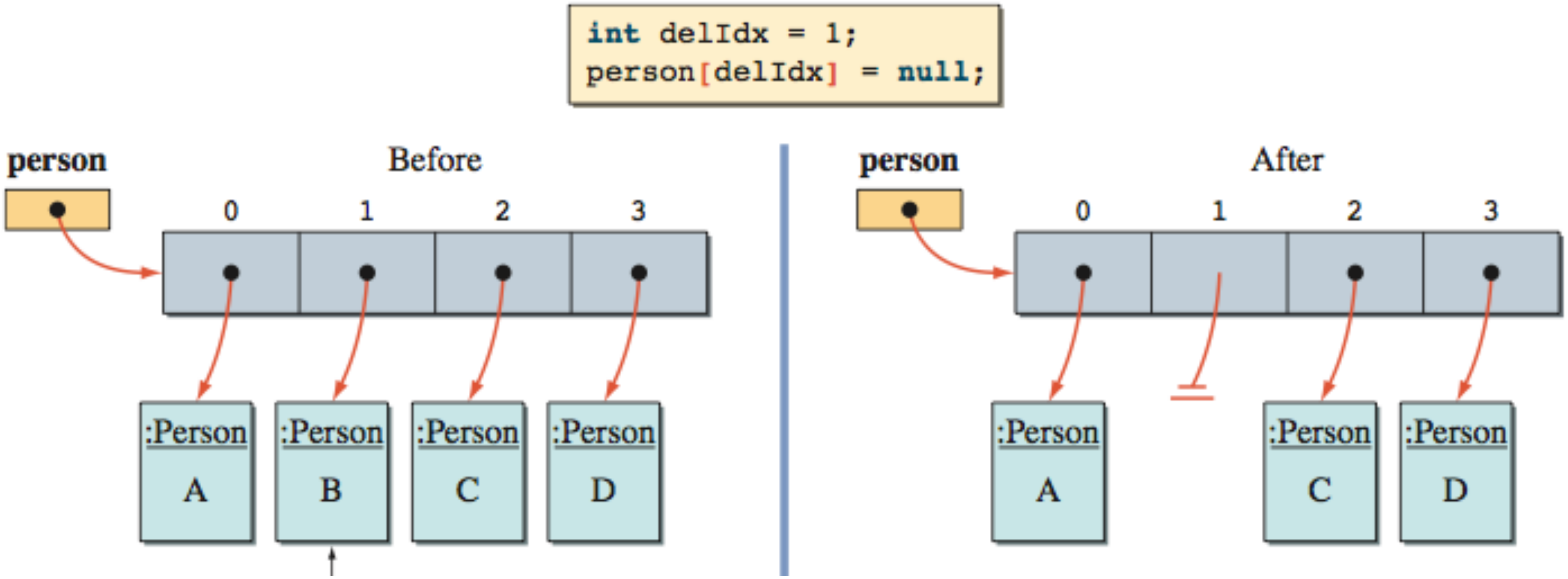
- To search for a particular person. We can scan through the array until the desired person is found.



```java
int i = 0;

while (i < person.length &&//still more persons to search
        !person[i].getName().equals("Latte")) {
    i++;
}

if (i == person.length) {
    //not found - unsuccessful search
    System.out.println("Ms. Latte was not in the array");

} else {
    //found - successful search
    System.out.println("Found Ms. Latte at position " + i);
}
```

Arrays of Objects

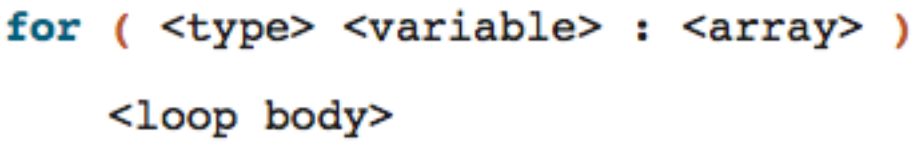
- To delete object from array could be accomplished by setting the reference to null.



```
int delIdx = 1;
person[delIdx] = null;
```

The For-Each Loop

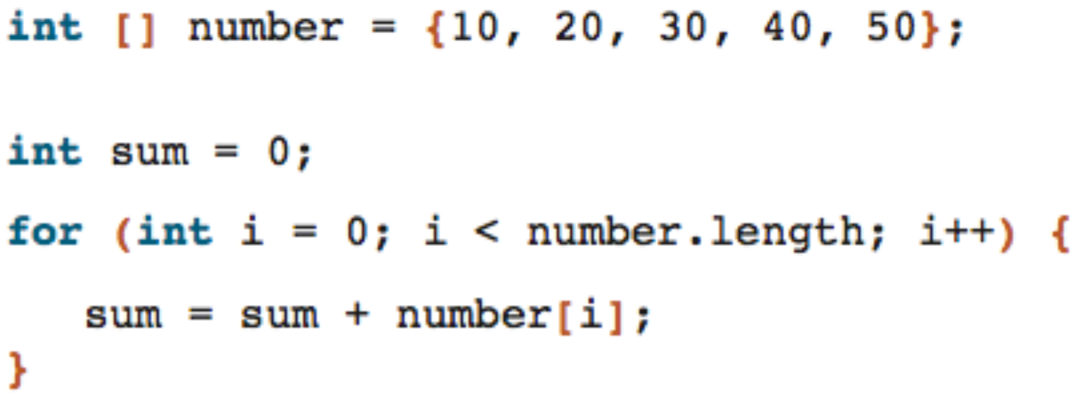
- The loop iterates over every element in the array, and the loop body is executed for each iteration. We can interpret this loop as saying something like “For each value in array, execute the following loop body.”. The general syntax for the for-each loop is



```
for ( <type> <variable> : <array> )

     <loop body>
```

The For-Each Loop

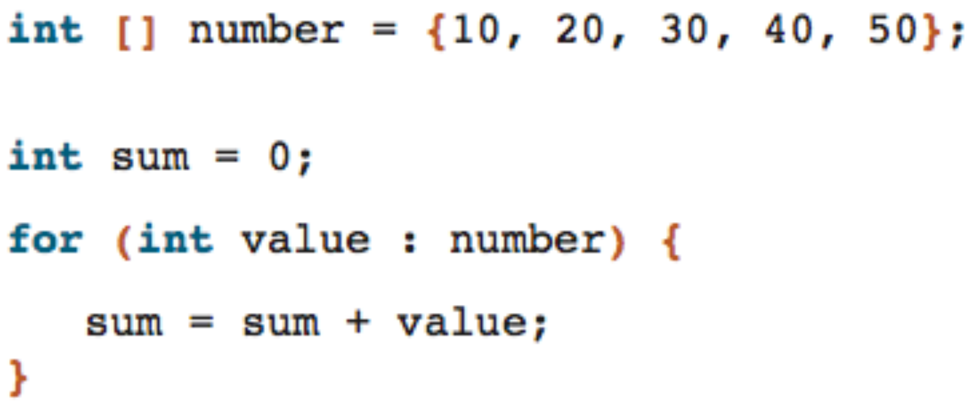
- Let’s assume number is an array of integers. To compute the sum of all elements in the number array:
    - Using the standard for loop



```
int [] number = {10, 20, 30, 40, 50};

int sum = 0;
for (int i = 0; i < number.length; i++) {
    sum = sum + number[i];
}
```

The For-Each Loop

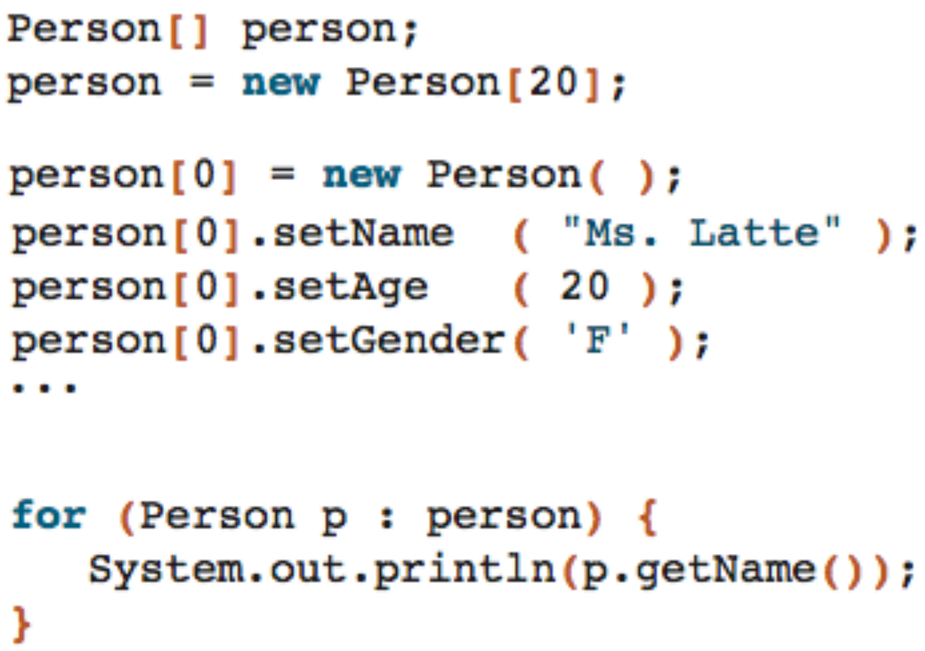
– Using a for-each loop



```
int [] number = {10, 20, 30, 40, 50};

int sum = 0;
for (int value : number) {
   sum = sum + value;
}
```

The For-Each Loop

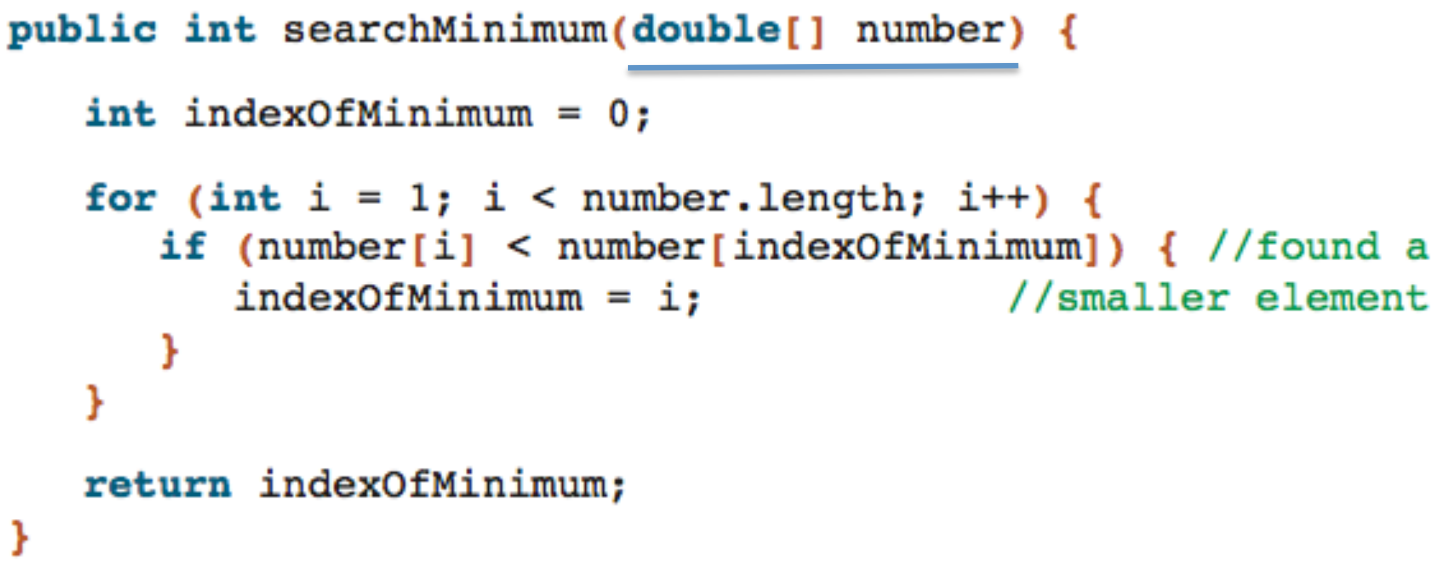
- Let’s assume person is an array of Person objects. To display name of person in the array using for-each loop



```
Person[] person;
person = new Person[20];

person[0] = new Person( );
person[0].setName   ( "Ms. Latte" );
person[0].setAge    ( 20 );
person[0].setGender( 'F' );
...


for (Person p : person) {
   System.out.println(p.getName());
}
```

Passing Arrays to Methods

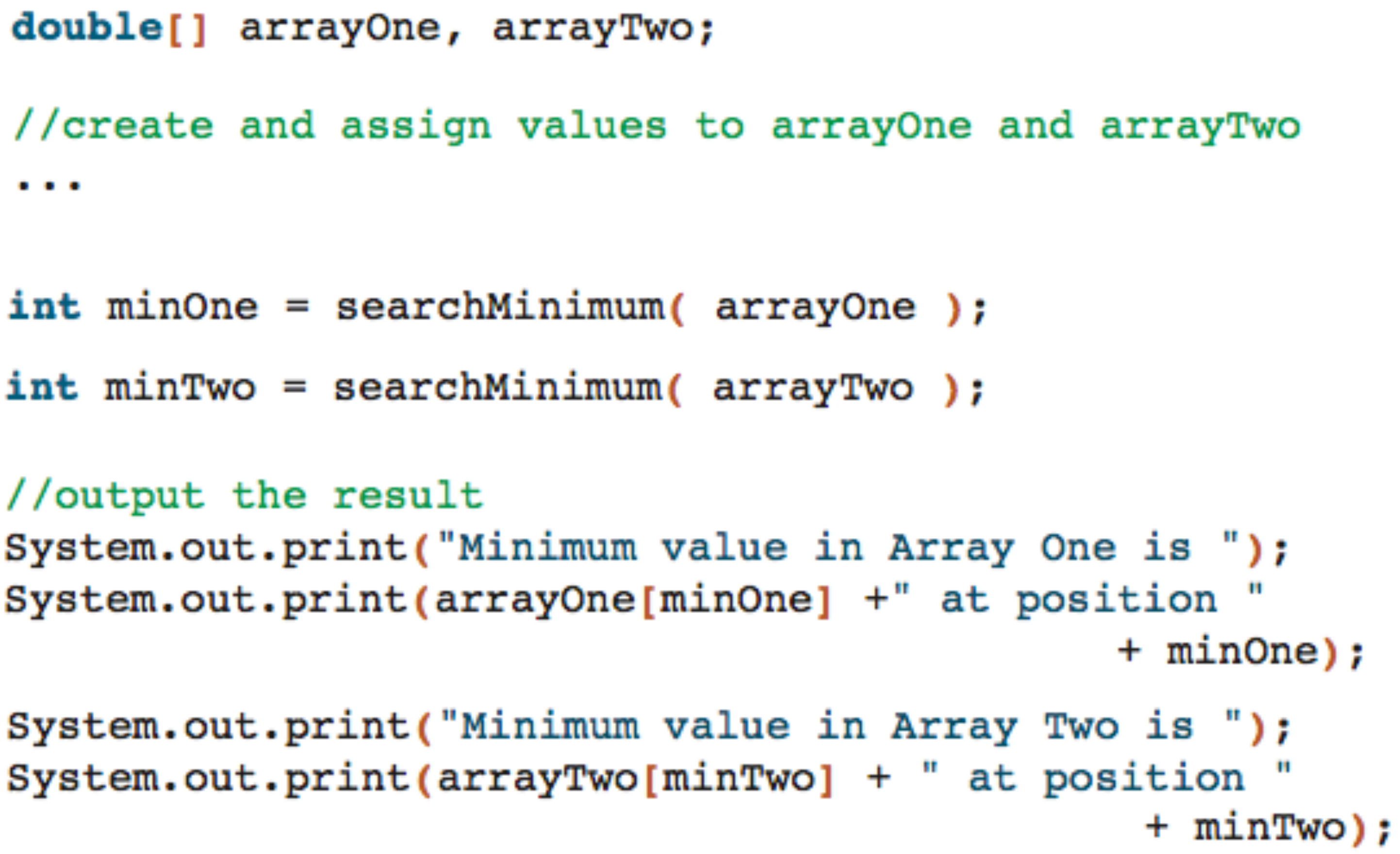
- Let’s define a method that returns the index of the smallest element in an array of real numbers. The array to search for the smallest element is passed to the method.



```
public int searchMinimum(double[] number) {

    int indexOfMinimum = 0;

    for (int i = 1; i < number.length; i++) {
        if (number[i] < number[indexOfMinimum]) { //found a
            indexOfMinimum = i;                 //smaller element
        }
    }

    return indexOfMinimum;
}
```

Passing Arrays to Methods

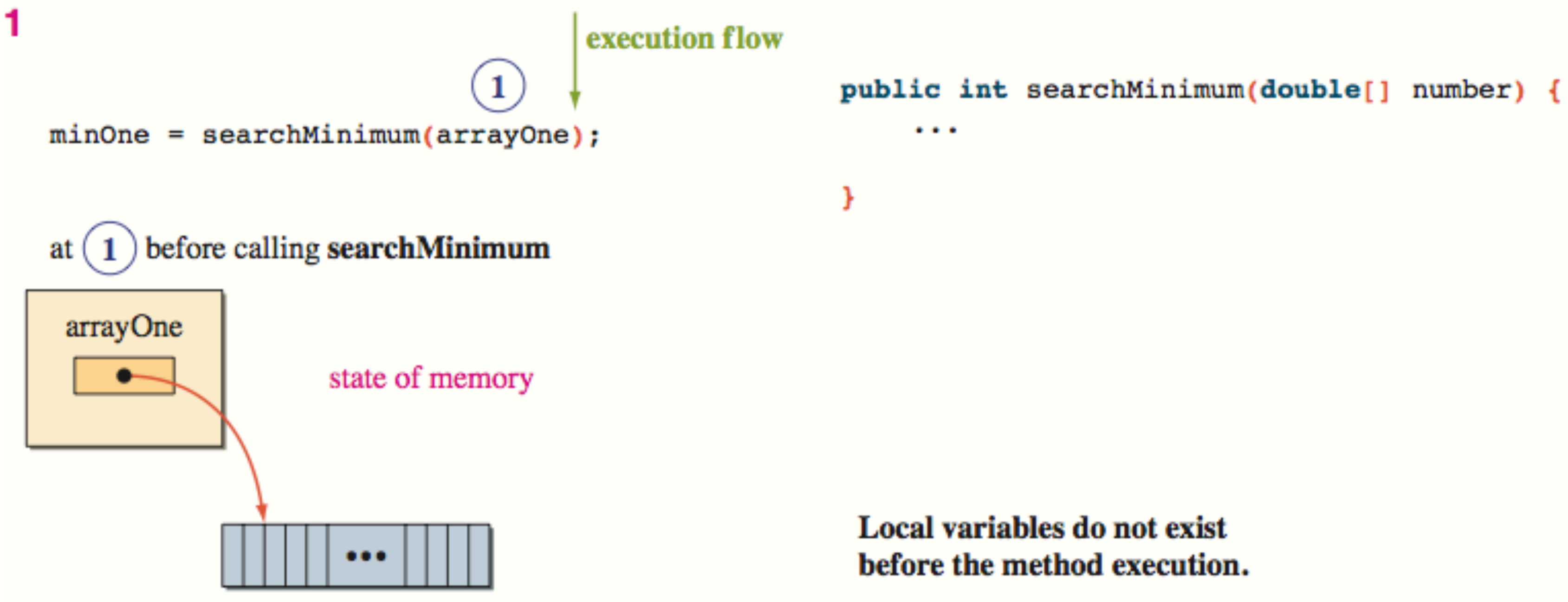
– To call this method, we write something like this:



```
double[] arrayOne, arrayTwo;

//create and assign values to arrayOne and arrayTwo
...

int minOne = searchMinimum( arrayOne );

int minTwo = searchMinimum( arrayTwo );

//output the result
System.out.print("Minimum value in Array One is ");
System.out.print(arrayOne[minOne] +" at position "
                                              + minOne);

System.out.print("Minimum value in Array Two is ");
System.out.print(arrayTwo[minTwo] + " at position "
                                               + minTwo);
```

Passing Arrays to Methods

– An array is a reference data type, so we are passing the reference to an array, not the whole array, when we call the searchMinimum() method.

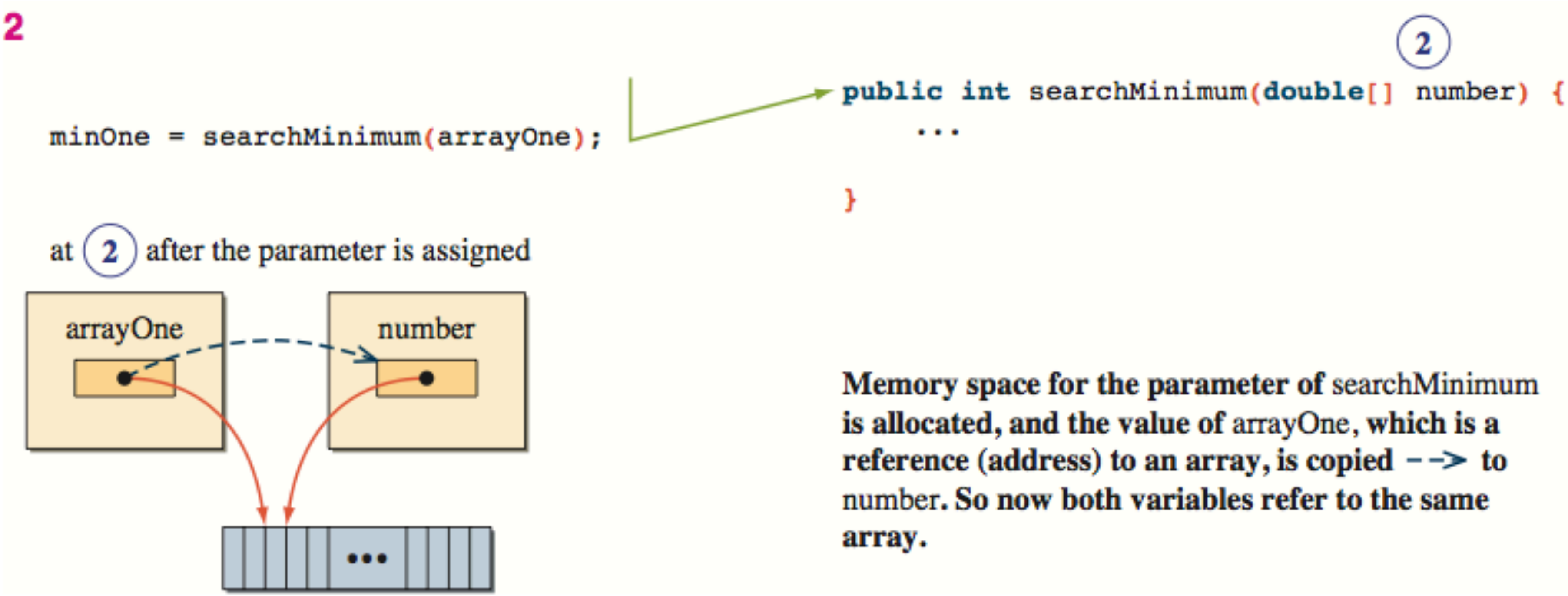


```
minOne = searchMinimum(arrayOne);
```

```
public int searchMinimum(double[] number) {
    . . .
}
```

Local variables do not exist before the method execution.

Passing Arrays to Methods



2

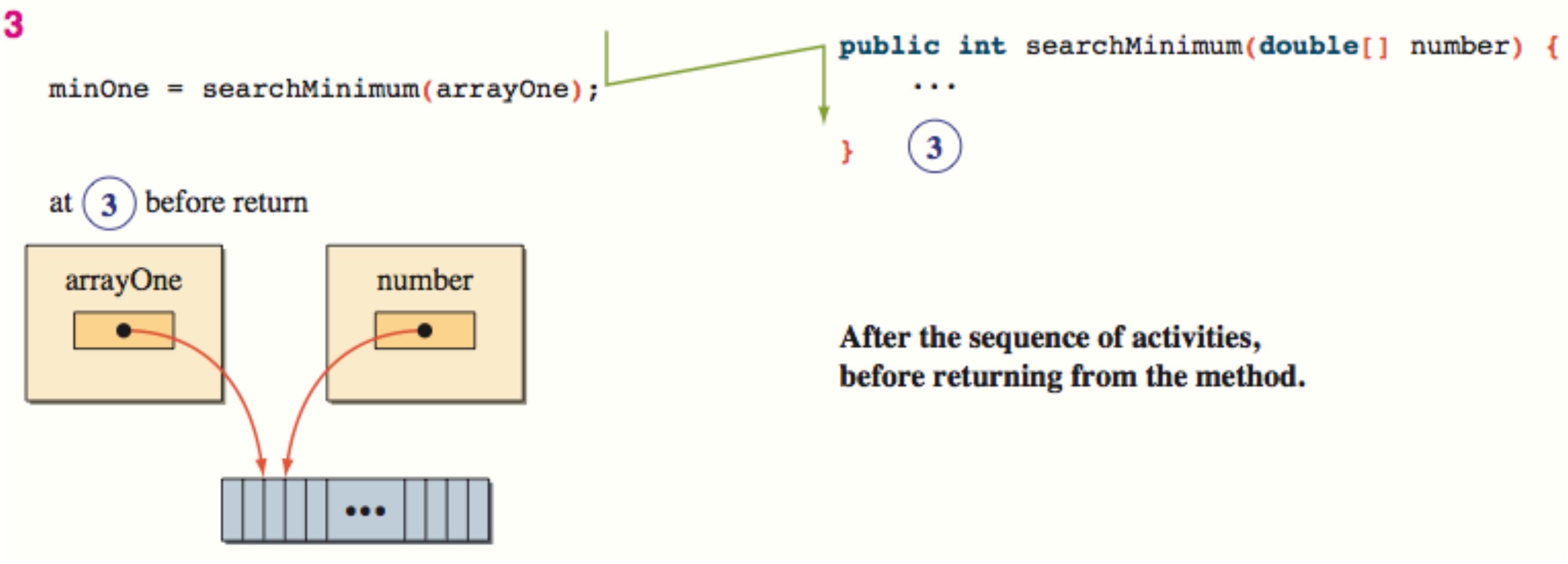
```
minOne = searchMinimum(arrayOne);
```

```
public int searchMinimum(double[] number) {
    ...

}
```

at (2) after the parameter is assigned

Memory space for the parameter of searchMinimum is allocated, and the value of arrayOne, which is a reference (address) to an array, is copied --> to number. So now both variables refer to the same array.

Passing Arrays to Methods

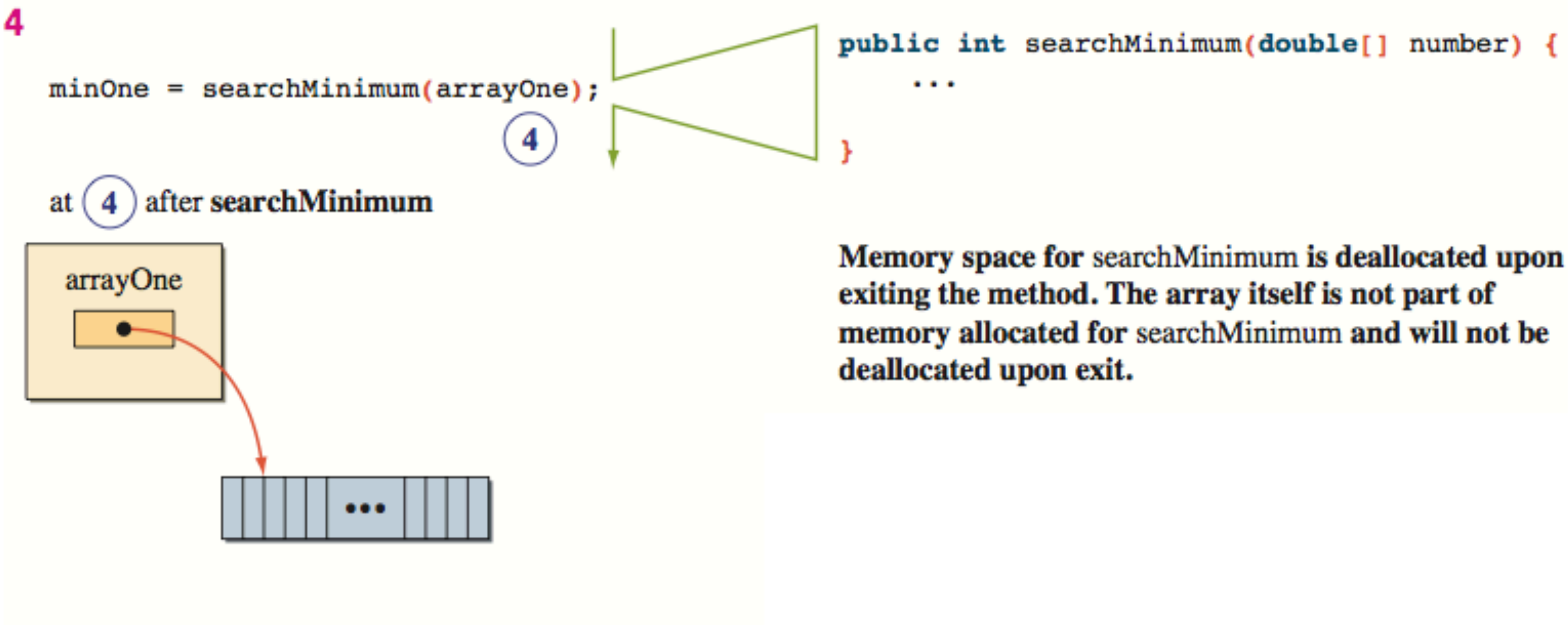


```
minOne = searchMinimum(arrayOne);
```

```
public int searchMinimum(double[] number) {
    . . .
}
```

After the sequence of activities, before returning from the method.

Passing Arrays to Methods



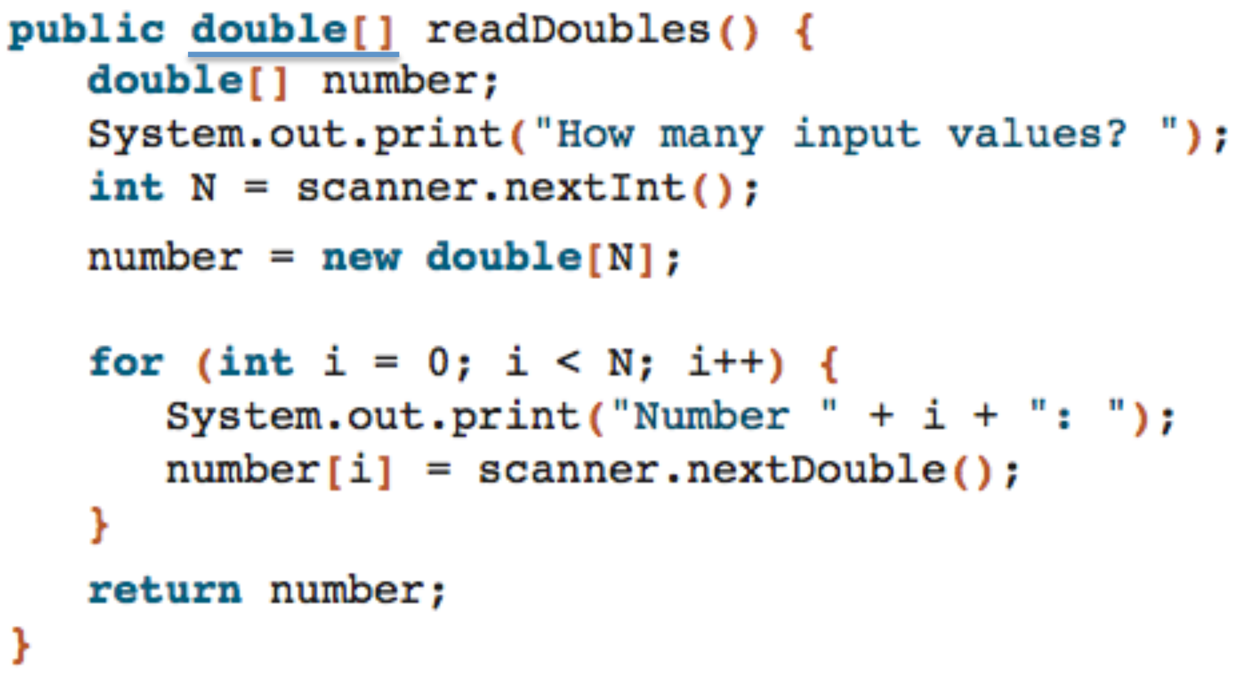
```
minOne = searchMinimum(arrayOne);
```

```
public int searchMinimum(double[] number) {
    ...
}
```

Memory space for searchMinimum is deallocated upon exiting the method. The array itself is not part of memory allocated for searchMinimum and will not be deallocated upon exit.

Passing Arrays to Methods

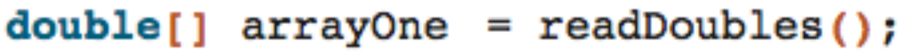
- Now let’s try another example in which we return an array from a method.



```
public double[] readDoubles() {
    double[] number;
    System.out.print("How many input values? ");
    int N = scanner.nextInt();
    number = new double[N];

    for (int i = 0; i < N; i++) {
        System.out.print("Number " + i + ": ");
        number[i] = scanner.nextDouble();
    }
    return number;
}
```

Passing Arrays to Methods

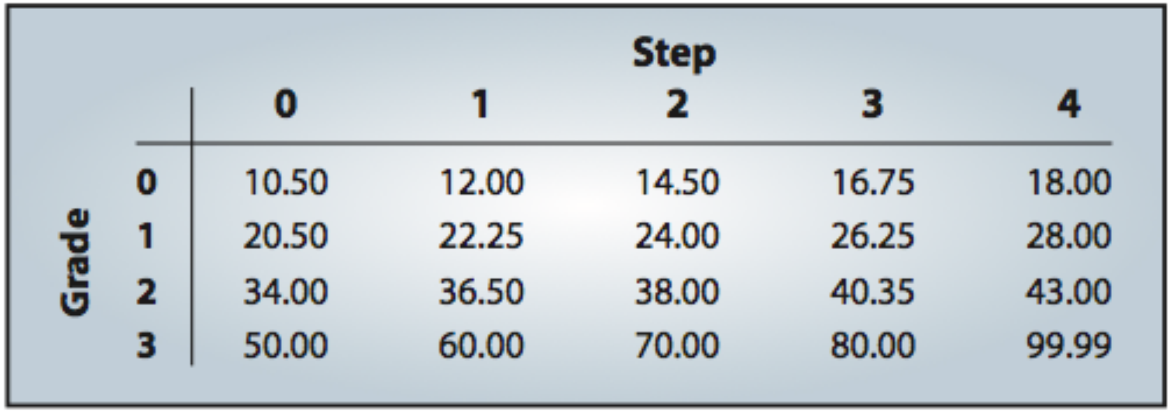
– The readDoubles() method is called in this manner:



```
double[] arrayOne = readDoubles();
```

Two-Dimensional Arrays

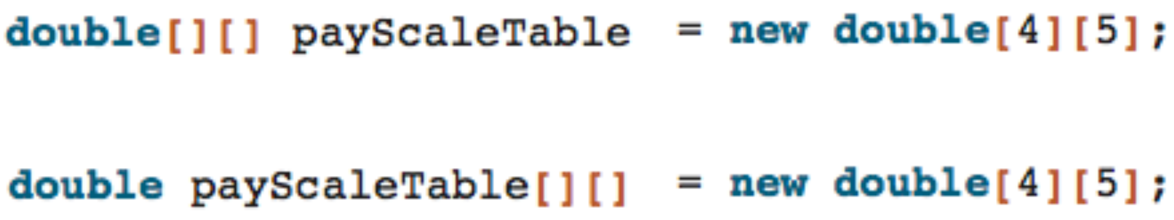
- In Java, we represent tables as two-dimensional arrays. The table contains the hourly rate of programmers based on their skill level. The rows represent the grade levels, the columns represent the steps.



				Step		
		0	1	2	3	4
Grade	0	10.50	12.00	14.50	16.75	18.00
	1	20.50	22.25	24.00	26.25	28.00
	2	34.00	36.50	38.00	40.35	43.00
	3	50.00	60.00	70.00	80.00	99.99

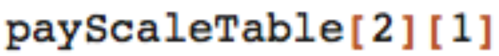
Two-Dimensional Arrays

– We declare the pay scale table as



```
double[][] payScaleTable  = new double[4][5];

double payScaleTable[][]  = new double[4][5];
```

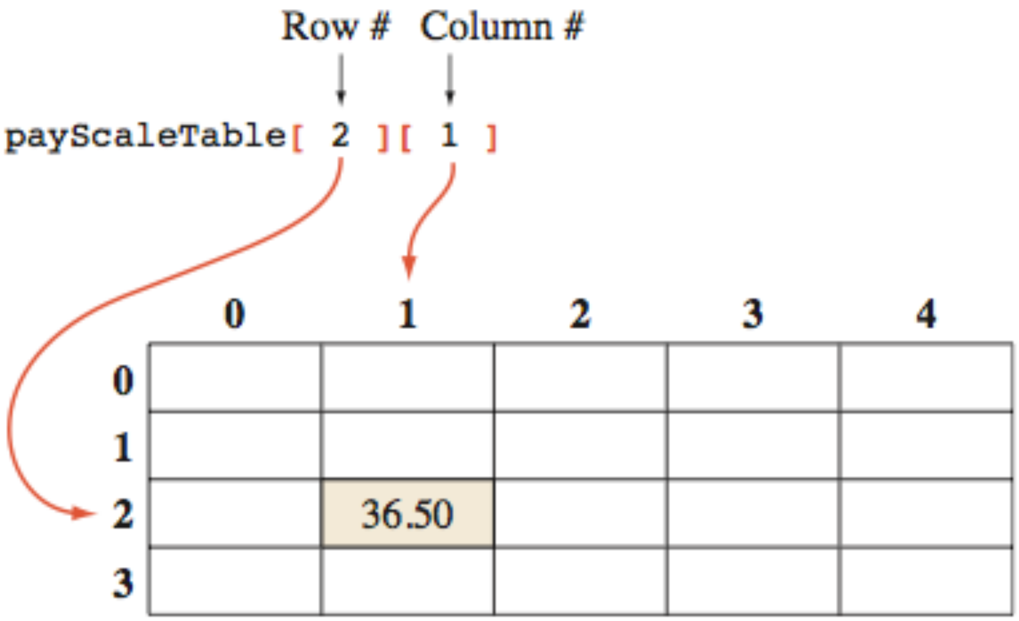
– To refer to the element at the second column of the third row, we say



```
payScaleTable[2][1]
```

Two-Dimensional Arrays

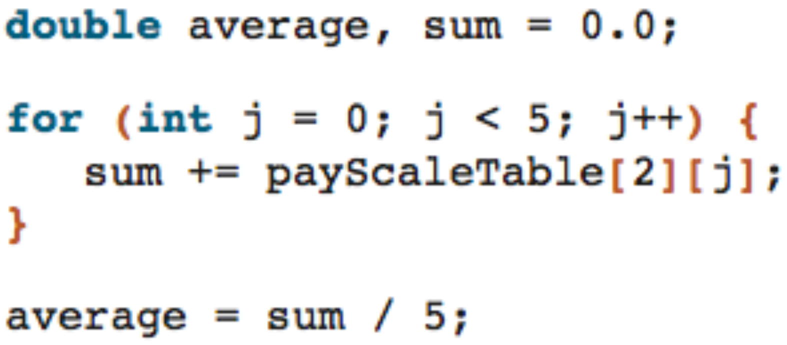
– Figure below illustrates how the two indices are used to access an array element of a two-dimensional array.



	0	1	2	3	4
0					
1					
2		36.50			
3					

Two-Dimensional Arrays

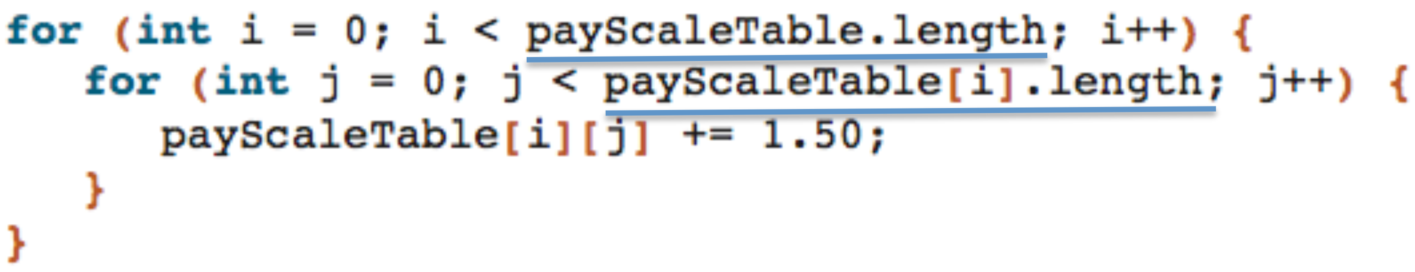
- Let’s go over some examples to see how the elements of two-dimensional arrays are manipulated.
  - This code finds the average pay of the grade 2 programmers.



```
double average, sum = 0.0;

for (int j = 0; j < 5; j++) {
    sum += payScaleTable[2][j];
}

average = sum / 5;
```

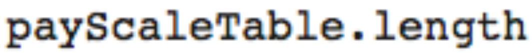
Two-Dimensional Arrays

– This code adds $1.50 to every skill level.

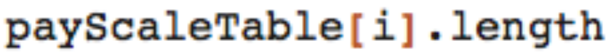


```
for (int i = 0; i < payScaleTable.length; i++) {
   for (int j = 0; j < payScaleTable[i].length; j++) {
      payScaleTable[i][j] += 1.50;
   }
}
```

– Notice a difference between 2 expressions.



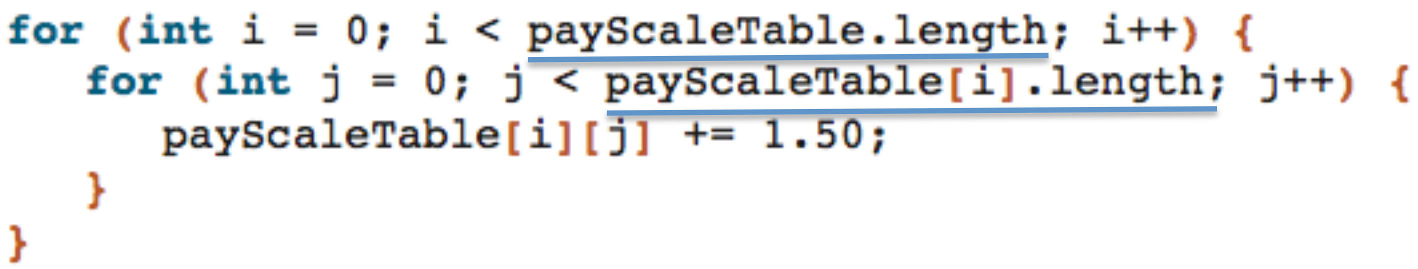
```
payScaleTable.length
```



```
payScaleTable[i].length
```

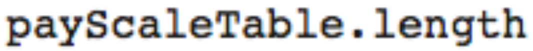
Two-Dimensional Arrays

– This code adds $1.50 to every skill level.

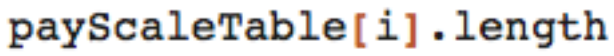


```
for (int i = 0; i < payScaleTable.length; i++) {
    for (int j = 0; j < payScaleTable[i].length; j++) {
        payScaleTable[i][j] += 1.50;
    }
}
```

– Notice a difference between 2 expressions.



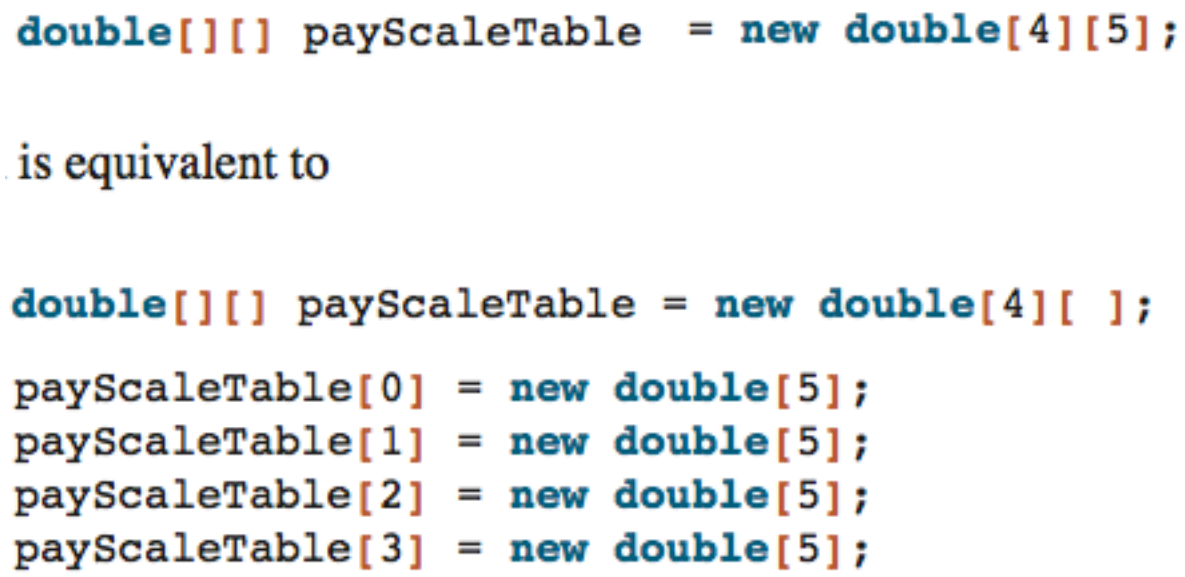
```
payScaleTable.length
```



```
payScaleTable[i].length
```

Two-Dimensional Arrays

- There is actually no explicit structure called two-dimensional array in Java. We only have an array of arrays.
  - The two-dimensional array creation



```
double[][] payScaleTable  = new double[4][5];
```

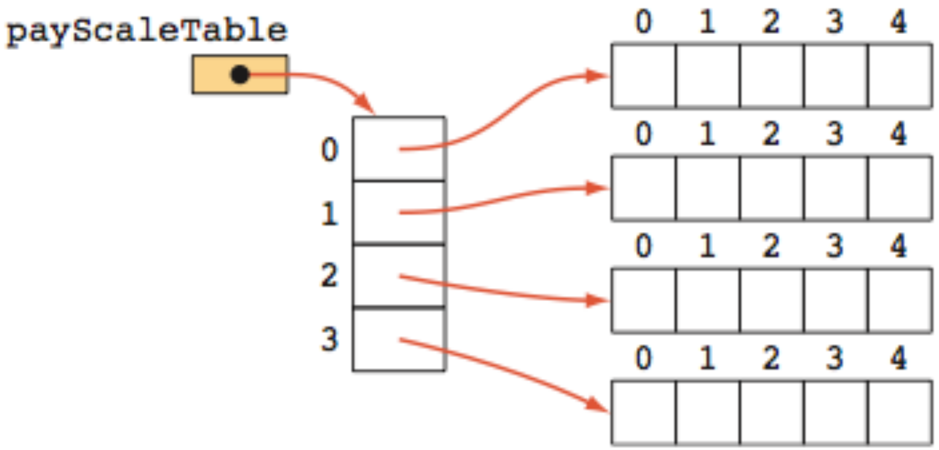
is equivalent to

```
double[][] payScaleTable = new double[4][ ];

payScaleTable[0] = new double[5];
payScaleTable[1] = new double[5];
payScaleTable[2] = new double[5];
payScaleTable[3] = new double[5];
```

Two-Dimensional Arrays

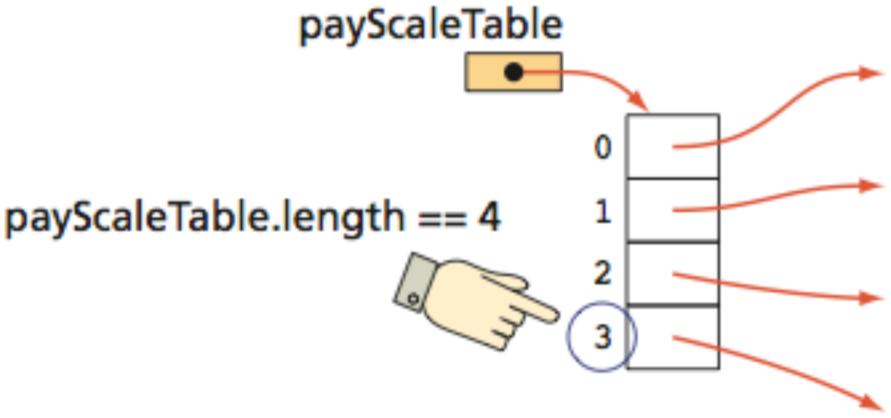
– Figure below shows the effect of executing the statements.

```
double[][] payScaleTable  = new double[4][5];
```



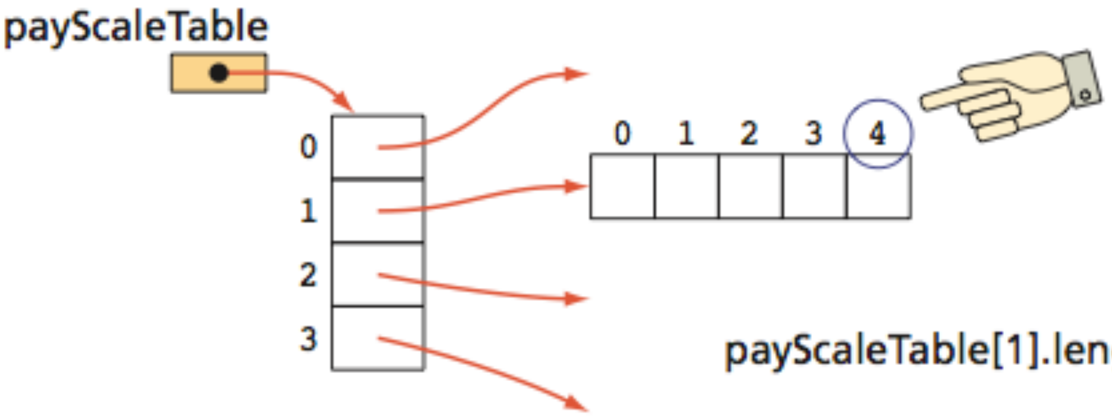
Two-Dimensional Arrays

– The expression payScaleTable.length refers to the length of the payScaleTable array itself.



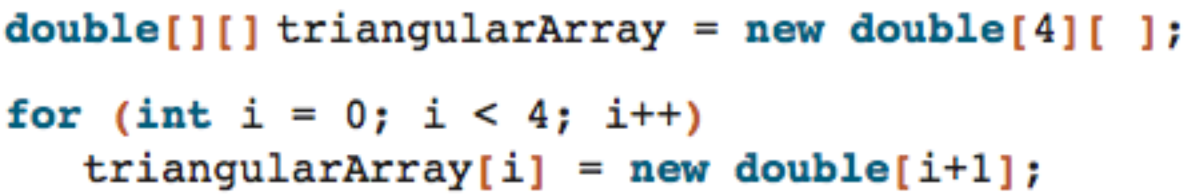
Two-Dimensional Arrays

– And the expression payScaleTable[1].length refers to the length of an array stored at row 1 of payScaleTable.

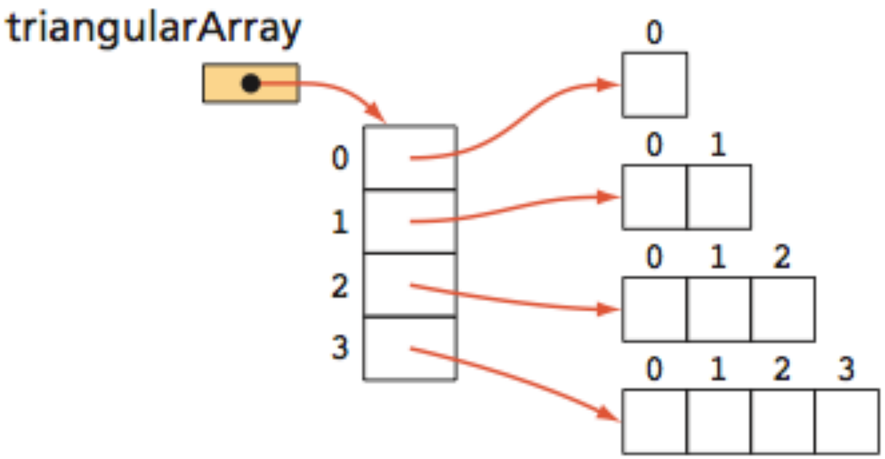


Two-Dimensional Arrays

– Since we allocate the subarrays individually, we can create subarrays of different lengths.

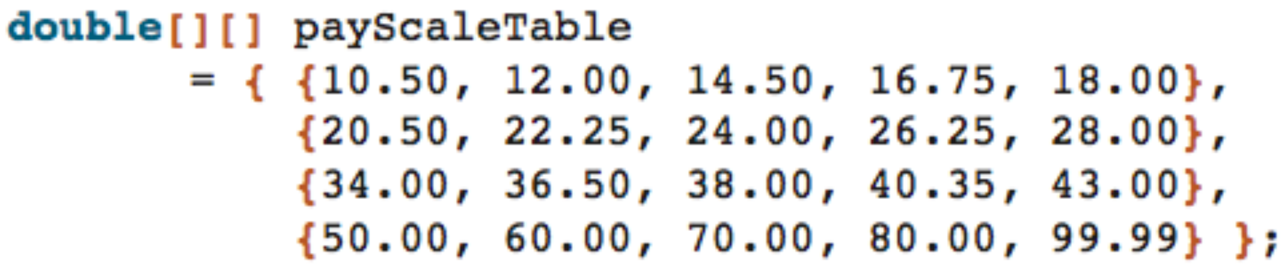


```
double[][] triangularArray = new double[4][ ];

for (int i = 0; i < 4; i++)
   triangularArray[i] = new double[i+1];
```



Two-Dimensional Arrays

– An array of arrays can be initialized at the time of declaration. The following declaration initializes the payScaleTable array:



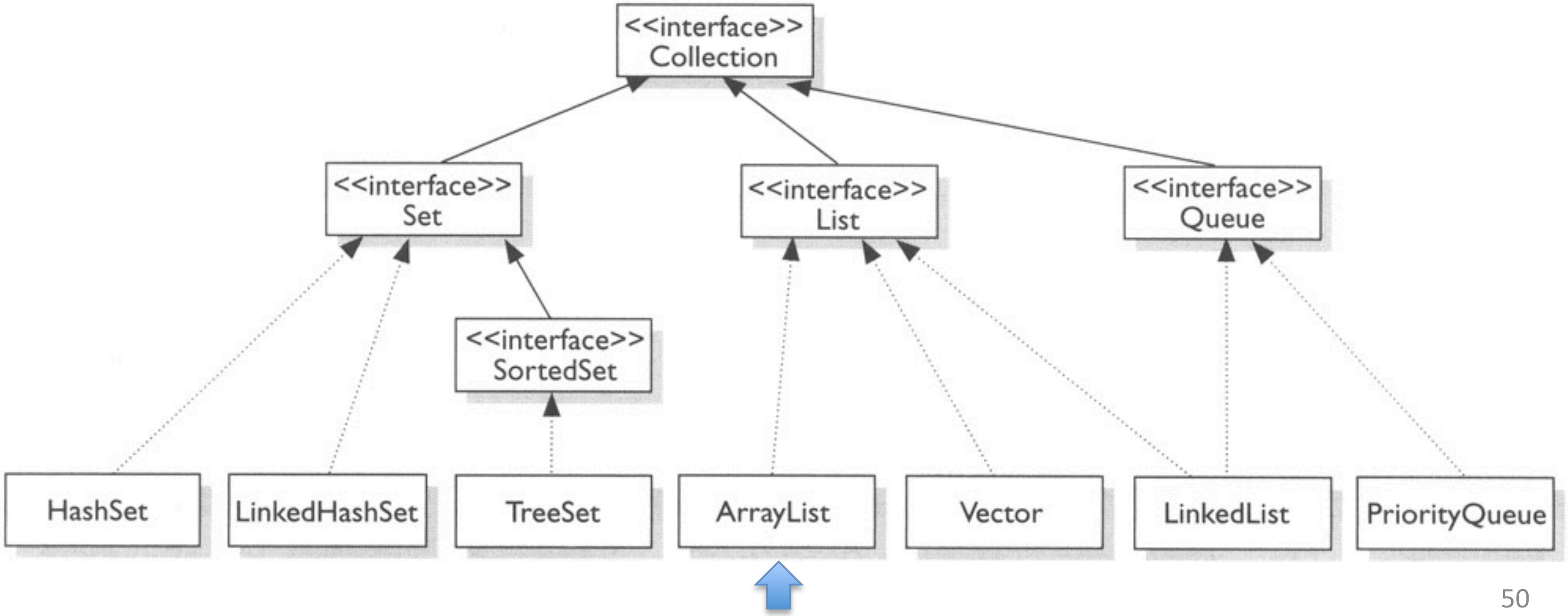
```
double[][] payScaleTable
        = { {10.50, 12.00, 14.50, 16.75, 18.00},
            {20.50, 22.25, 24.00, 26.25, 28.00},
            {34.00, 36.50, 38.00, 40.35, 43.00},
            {50.00, 60.00, 70.00, 80.00, 99.99} };
```

Lists and Maps

- In Java standard library "java.util" already includes various classes and interfaces for maintaining a collection of objects. They are collectively referred as the Java Collection Framework, or JCF. We will study Lists and Maps in this section.

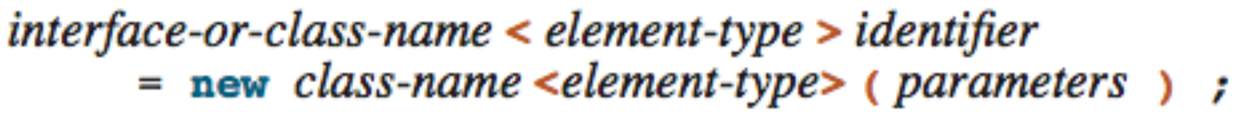
Lists

- The first is the List interface. There are two classes in JCF that implement the List interface: ArrayList and LinkedList.

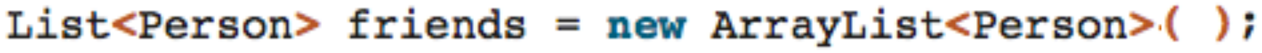


Lists

- The ArrayList class uses as an array. By using this class, we can keep adding new elements without worrying about the array size.
- Let’s use the ArrayList class. The general syntax for the declaration is



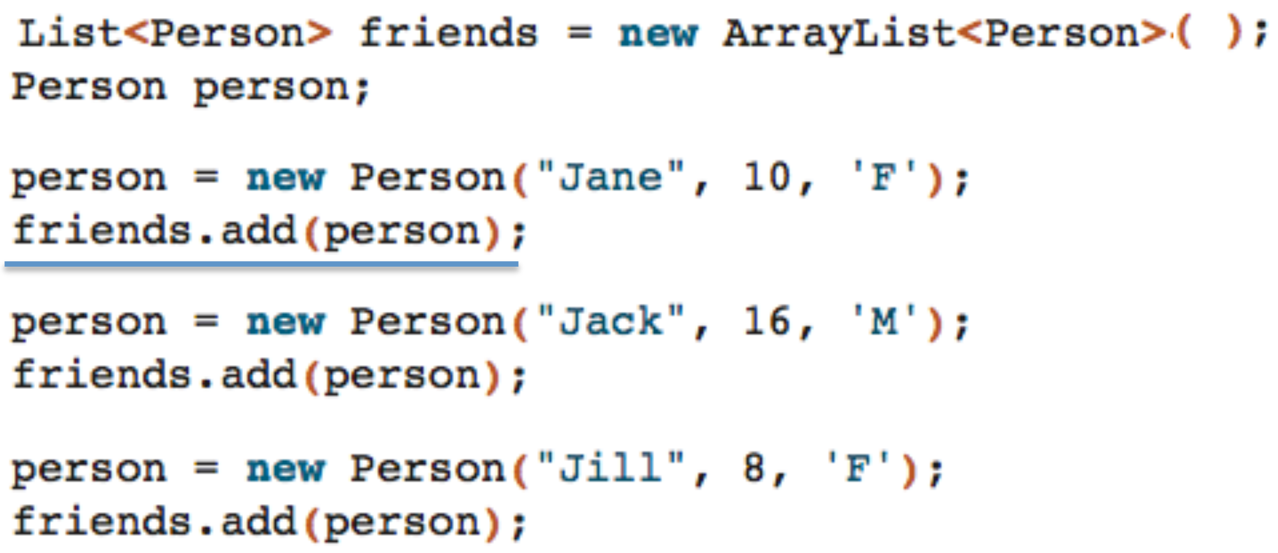
```
interface-or-class-name < element-type > identifier
        = new class-name <element-type> ( parameters ) ;
```



```
List<Person> friends = new ArrayList<Person>( );
```

Lists

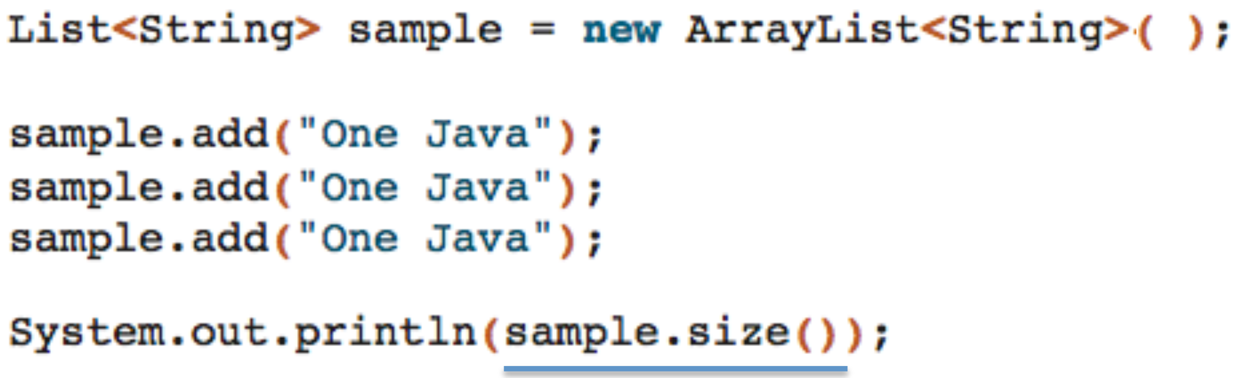
- Now we are ready to study the basic operations of the List interface.
  - add(): we use add() method to add object to the list



```
List<Person> friends = new ArrayList<Person>( );
Person person;

person = new Person("Jane", 10, 'F');
friends.add(person);

person = new Person("Jack", 16, 'M');
friends.add(person);

person = new Person("Jill", 8, 'F');
friends.add(person);
```

Lists

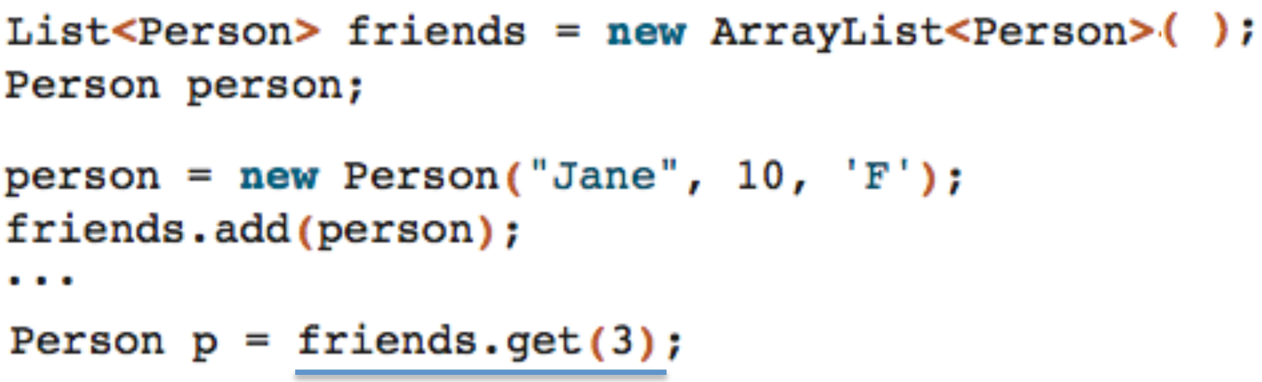
– size(): we use size() method to get number of object in the list



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

sample.add("One Java");
sample.add("One Java");
sample.add("One Java");

System.out.println(sample.size());
```

Lists

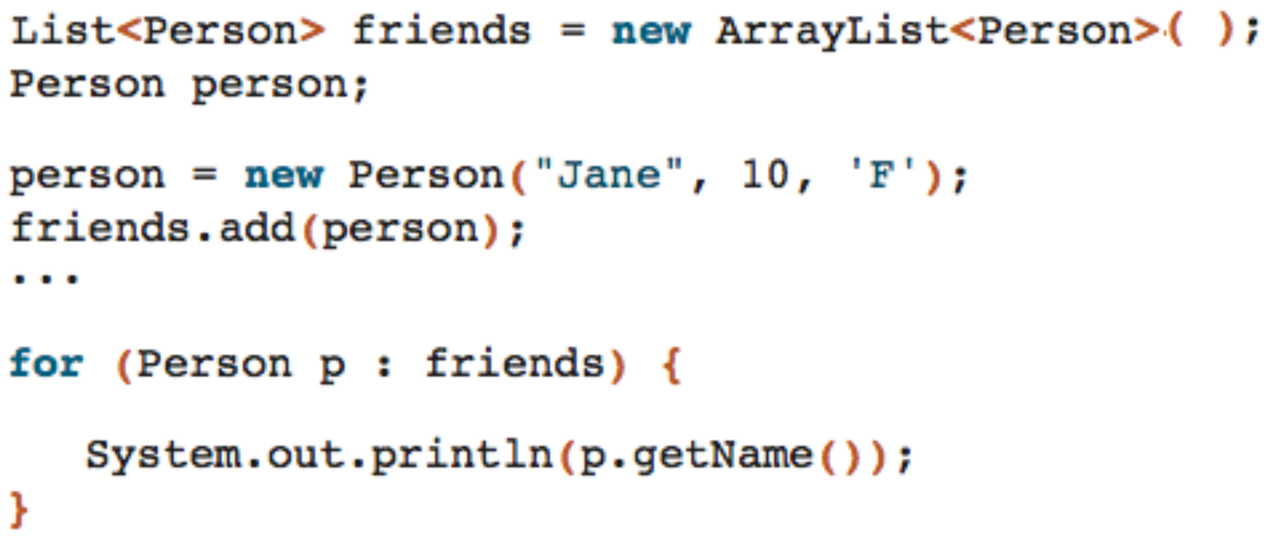
– get(): we use the get() method to access an object at index position i. For example, to access the Person object at position 3 (the 4th element). An invalid argument, such as a negative value or a value greater than size() – 1, will result in an IndexOutOfBoundsException error.



```
List<Person> friends = new ArrayList<Person>( );
Person person;

person = new Person("Jane", 10, 'F');
friends.add(person);
...
Person p = friends.get(3);
```

Lists

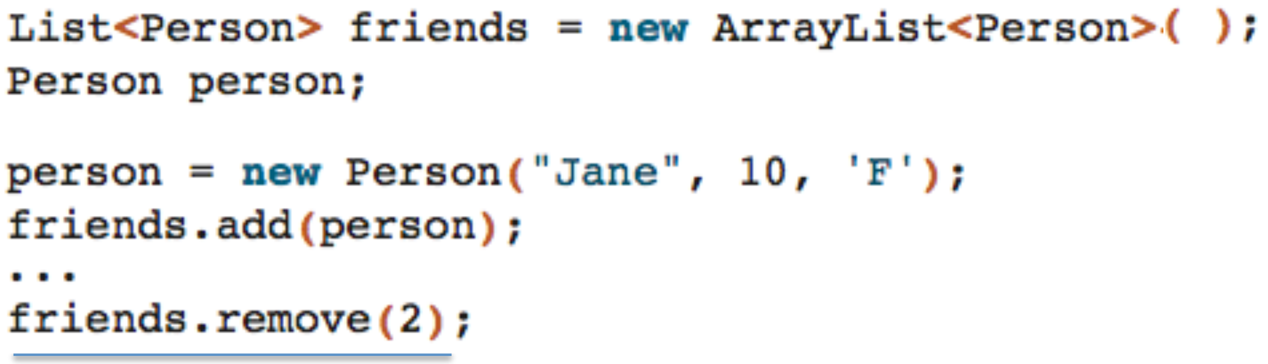
– To traverse a list from the first to the last element, we can use the for-each loop.



```
List<Person> friends = new ArrayList<Person>( );
Person person;

person = new Person("Jane", 10, 'F');
friends.add(person);
...

for (Person p : friends) {

    System.out.println(p.getName());
}
```

Lists

– remove(): we use the remove() method to remove an element at index i from a list.



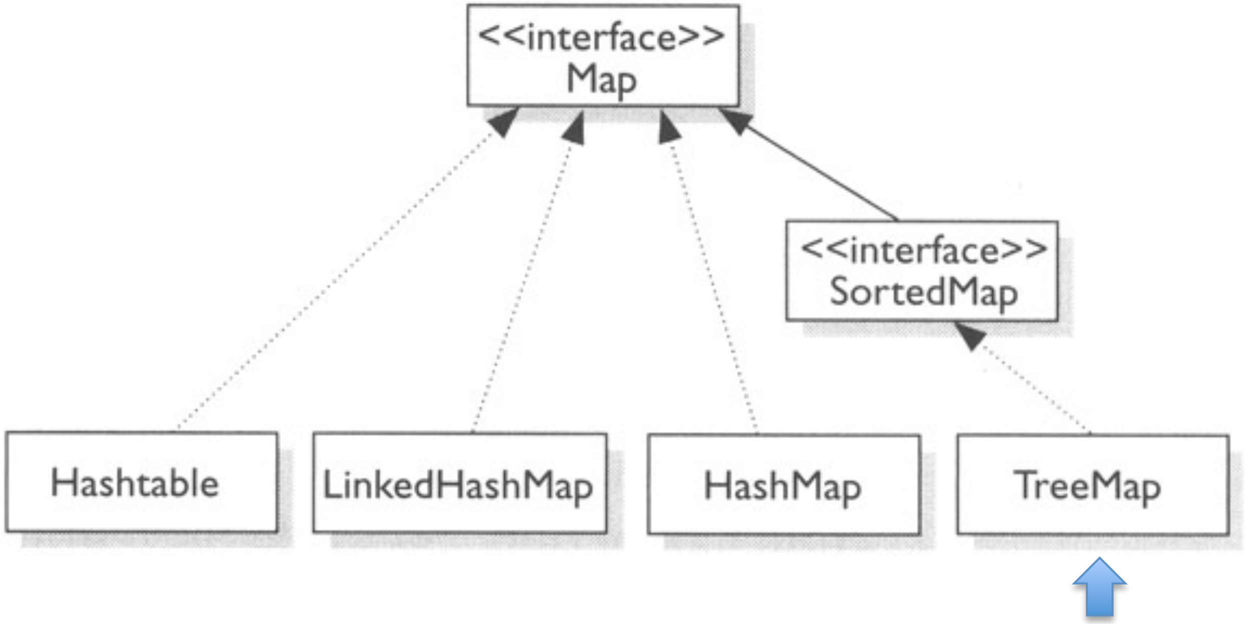
```
List<Person> friends = new ArrayList<Person>( );
Person person;

person = new Person("Jane", 10, 'F');
friends.add(person);
...
friends.remove(2);
```

Maps

- A map consists of entries, with each entry has two parts: key and value. No duplicate keys are allowed in the map. Both key and value can be an instance of any class.

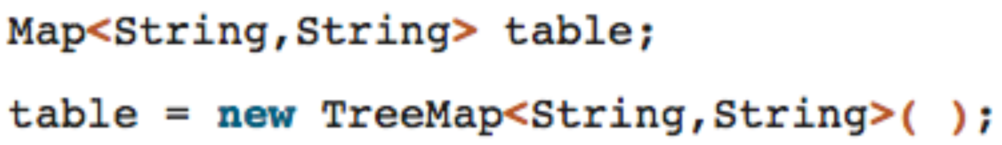
Maps

- There are two classes that implement this interface: HashMap and TreeMap. We will describe the TreeMap class in this section.



Maps

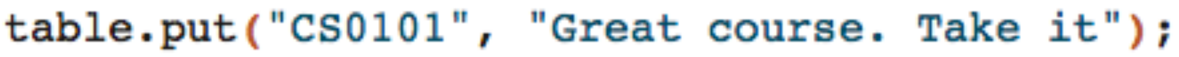
- Now we are ready to study the basic operations of the List interface.
  - To declare and create a map with String as both its key and value, we write



```
Map<String,String> table;

table = new TreeMap<String,String>( );
```

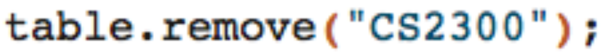
Maps

– put(): we use its put() method to add the key-value pairs to the map as



```
table.put("CS0101", "Great course. Take it");
```

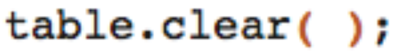
– remove(): to remove an entry, we use the remove() method with the key of an entry to remove from the map.



```
table.remove("CS2300");
```

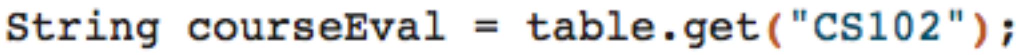
Maps

– clear(): we can remove all of them at once by calling the clear() method.



```
table.clear( );
```

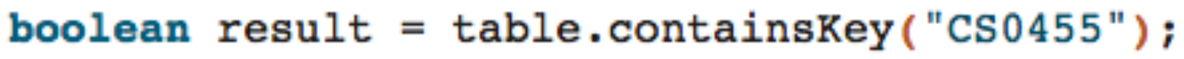
– get(): To retrieve the value associated to a key, we call get() method.



```
String courseEval = table.get("CS102");
```

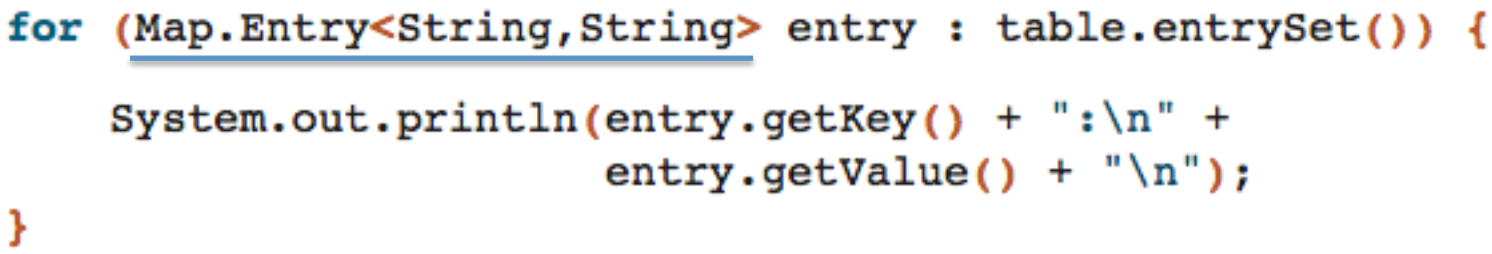
Maps

– containsKey(): to check the map contains specific key or not, we use contains() method.



```
boolean result = table.containsKey("CS0455");
```

– entrySet(): to get a set of elements (key and value), we use entrySet() method.



```
for (Map.Entry<String,String> entry : table.entrySet()) {
    System.out.println(entry.getKey() + ":\n" +
                       entry.getValue() + "\n");
}
```

Summary

- An array is an collection of data values.
- Individual elements in an array are accessed by the indexed expression.
- Array elements can be primitive data type values or objects.
- In Java, an array can include only elements of the same data type.

Summary

- A Java array is a reference data type.
- A Java array is created with the new operator.
- When an array is passed to a method as an argument, only a reference to an array is passed.

Summary

- The Java Collection Framework includes many data structure classes such as lists and maps.
- The List interface represents a linear ordered collection of objects.
- The Map interface represents a collection of key-value pairs.

Reference

- C. Thomas Wu, An Introduction to Object-Oriented Programming with Java, 5th Edition
  - Chapter 10: Arrays and Collections

Question?