Lecture 4:
Strings, Arrays and Dates

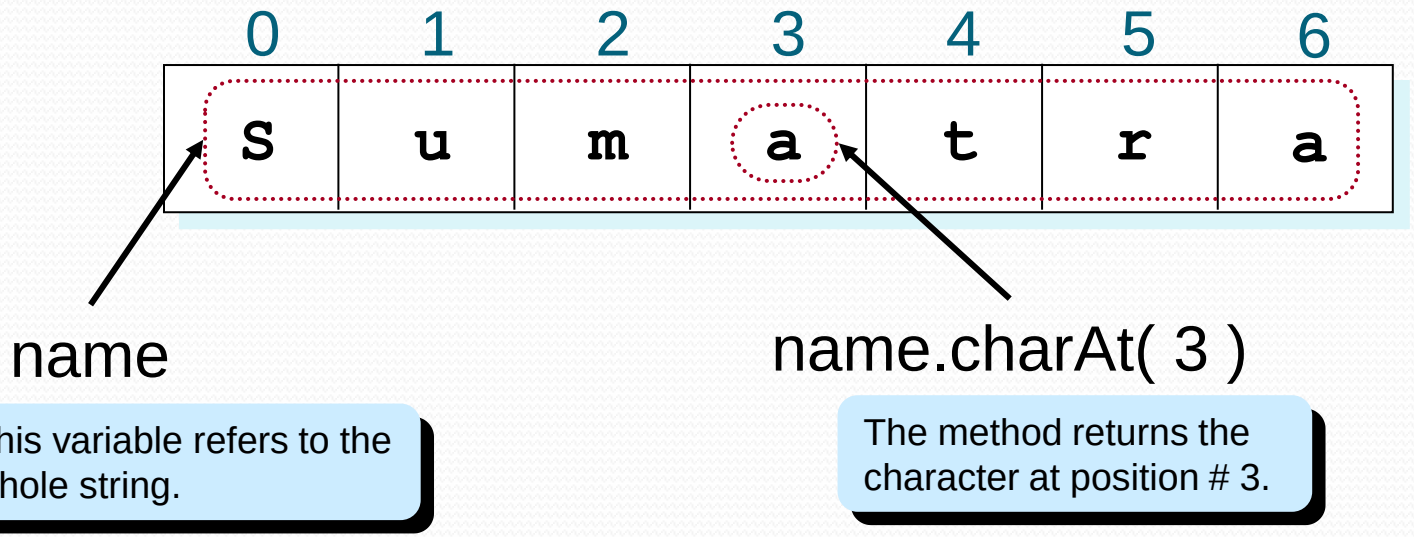
Strings

- A *string* is a sequence of characters that is treated as a single value.
- Instances of the **String** class are used to represent strings in Java.
- Two ways to create String objects
  - String s = “Hello”; //String literal
  - String t = new String(“World”);

Accessing Individual Elements

- Individual characters in a String accessed with the charAt method. Assume str is a String object, str.charAt(i) will return the character at the ith position. (0 <= i <= str.length() - 1)

```
String name = "Sumatra";
```



Comparing Objects: == and equals

- With primitive data types, we have only one way to compare them, but with objects (reference data type), we have two ways to compare them.
  1. We can test whether two variables point to the same object (use ==), or
  2. We can test whether two distinct objects have the same contents (use equals method).

Using == With String

```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

```
They are not equal
```

Not equal because str1 and str2 point to different String objects.

Using == With String

```
String str1 = new String("Java");
String str2 = str1;

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

```
They are equal
```

It's equal here because str1 and str2 point to the same object.

Using equals with String

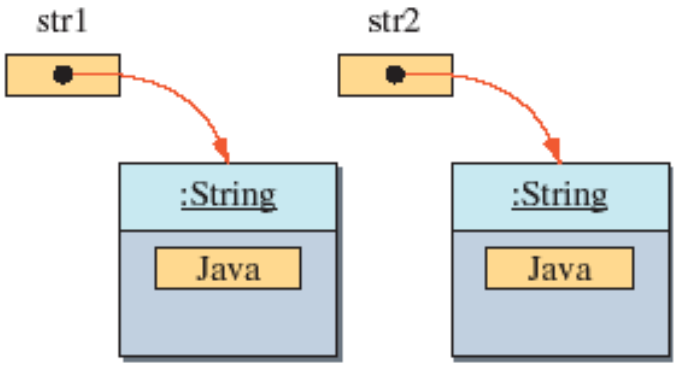
```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1.equals(str2)) {
     System.out.println("They are equal");
} else {
     System.out.println("They are not equal");
}
```

```
They are equal
```

It's equal here because str1 and str2 have the same sequence of characters.

The Semantics of ==

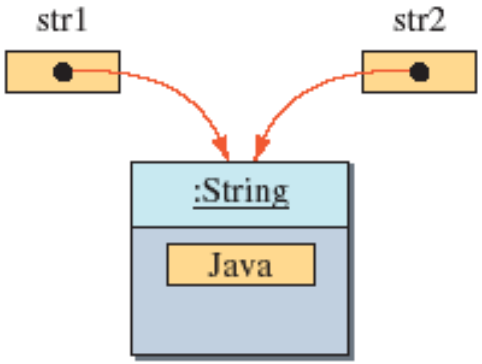
Case A: Two variables refer to two different objects.




```
String str1, str2;

str1 = new String("Java");
str2 = new String("Java");
```

str1 == str2 ⟶ false

Case B: Two variables refer to the same object.



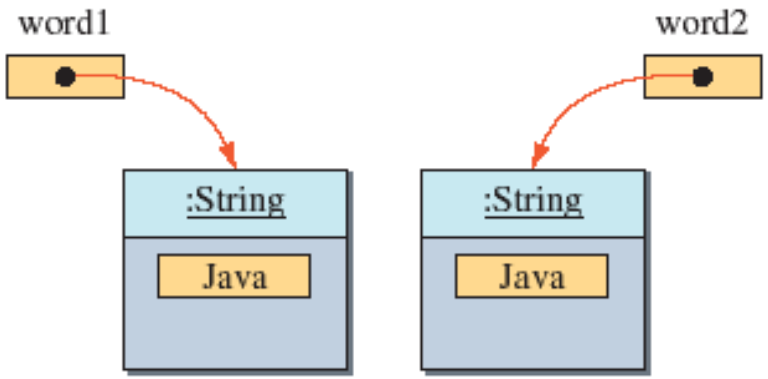

```
String str1, str2;

str1 = new String("Java");
str2 = str1;
```

str1 == str2 ⟶ true

In Creating String Objects

```
String word1, word2;

word1 = new String("Java");

word2 = new String("Java");
```

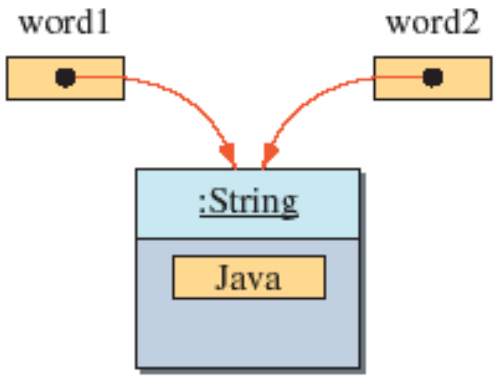
Whenever the **new** operator is used, there will be a new object.



word1 == word2 → false

```
String word1, word2;

word1 = "Java";

word2 = "Java";
```

Literal string constant such as "Java" will always refer to one object.



word1 == word2 → true

The String Class is Immutable

- In Java a String object is immutable
  - This means once a String object is created, it cannot be changed, such as replacing a character with another character or removing a character.
  - The String methods we have used so far do not change the original string. They created a new string from the original. For example, substring creates a new string from a given string.

Effect of Immutability

Question: Does the following code change the String object "java"?

```
String word1 = "java";
String word2 = word1.substring(0, 2);
String word3 = word1.replace('j', 'w');
```

String Functions: `compareTo`

- The natural ordering on Strings is *alphabetical order*. In that ordering, for example, "Bob" comes before "Charles". The `compareTo` method on `String`s specifies this ordering on `String`s:

```
int compareTo(String t)
```

- `s.compareTo(t)` returns
  - `a positive integer if s is "greater than" t`
  - `a negative integer is s is "less than" t`
  - `zero, if s and t are equal as Strings`
- Examples

```
System.out.println("a".compareTo("d"));
System.out.println("d".compareTo("a"));
System.out.println("a".compareTo("a"));
```

//output:

-3

3

0

String Functions: `substring`, `indexOf`

- Examples of how the `String` functions are used:
  - `substring`

```
String name = "Robert";
String nickname = name.substring(0,3);// “Rob”
String empty = name.substring(0,0);  // “”
String sub = name.substring(2); //”bert”
```

  - `indexOf`

```
String name = "Robert";
int posOfT = name.indexOf('t'); //5
int posOfSubstr = name.indexOf("bert"); //2
```

String Functions: startsWith,+

- startsWith

```
String name = "Robert";
boolean result = name.startsWith("Rob");//true
boolean result2 = name.startsWith("R"); //true
boolean result3 = name.startsWith("bert"); //false
```

- + (concatenation) – creates a new String

```
String name = "Robert";
String space = " ";
String lastName = "Stevens";
String fullname = name + space + lastName;
                        // "Robert Stevens"
```

Avoiding Costly Concatenation of Strings with StringBuilder

Problem: Concatenation(+) becomes very slow with many arguments because each concatenation creates a new `String` (which requires allocating new memory for the new object), and compared to other steps, this is a costly operation.

Solution: `StringBuilder` (demo: lesson4.stringbuilder)

StringBuilder represents a "growable String" – can append characters and `Strings` without significant cost.

Note: `StringBuilder` is designed to be used for single-threaded applications – it is not thread-safe. This means that a single `StringBuilder` instance must not be shared between two or more competing threads. If multithreaded access is needed, a class with the same method names, `StringBuffer`, can be used, but it is less efficient in the single-threaded case.

Array Basics

- An array is a collection of data values.
- If your program needs to deal with 100 integers, 500 Account objects, 365 real numbers, etc., you will use an array.
- In Java, an array is an indexed collection of data values of the same type.

Arrays of Primitive Data Types

- Array Declaration

```
<data type>[]  <variable>
```

- Array Creation

```
<variable>  = new <data type> [ <size> ]
```

- Example

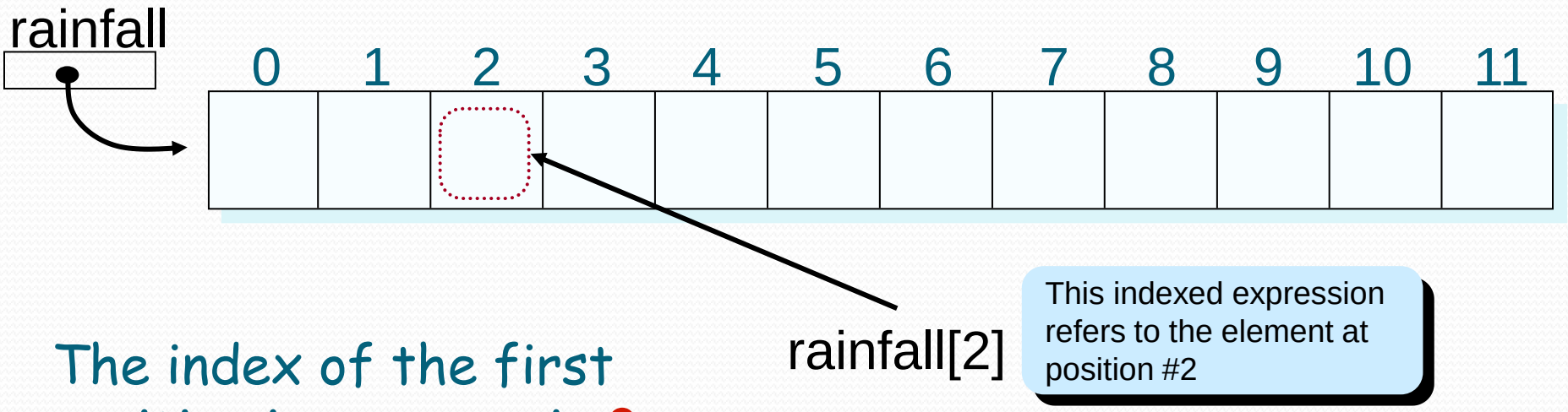
```
double[] rainfall;
rainfall
    = new double[12];
```

An array is like an object!

Accessing Individual Elements

- Individual elements in an array accessed with the indexed expression.

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



The index of the first position in an array is 0.

Array Initialization

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

double[] samplingData = { 2.443, 8.99, 12.3, 45.009, 18.2,
                          9.00, 3.123, 22.084, 18.08 };

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

number.length → **4**

samplingData.length → **9**

monthName.length → **12**

Arrays of Objects

- In Java, in addition to arrays of primitive data types, we can declare arrays of objects.
- We will use Person objects to illustrate the use of an array of objects.

```
Person latte = new Person();
latte.setName("Ms. Latte");
latte.setAge(20);
latte.setGender('F');
```

The Person class supports the set methods and get methods.

Creating an Object Array - 1

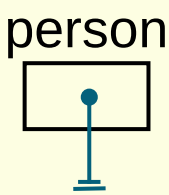
Code

A

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Only the name person is declared, no array is allocated yet.

State of Memory



After A is executed

Creating an Object Array - 2

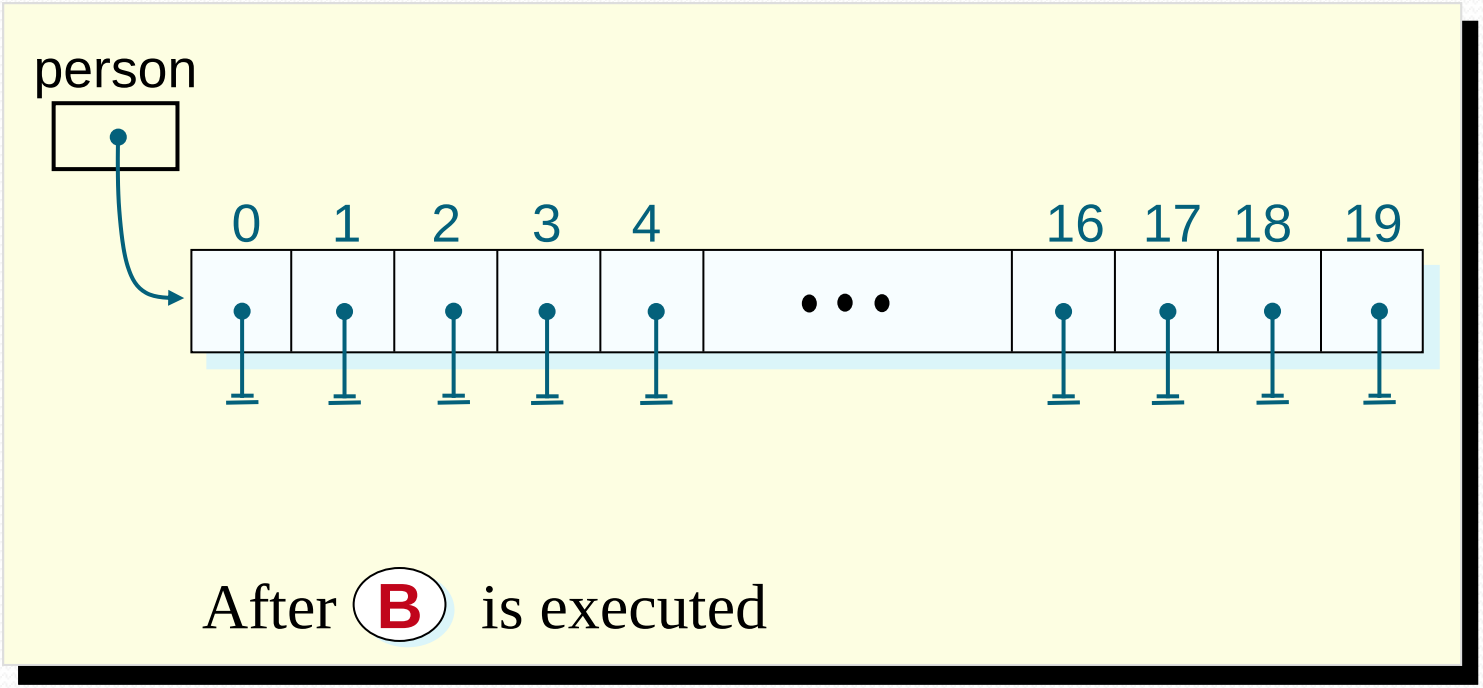
Code

B

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

Now the array for storing 20 Person objects is created and each element is **null**. The Person objects themselves are not yet created.

State of Memory



After B is executed

Creating an Object Array - 3

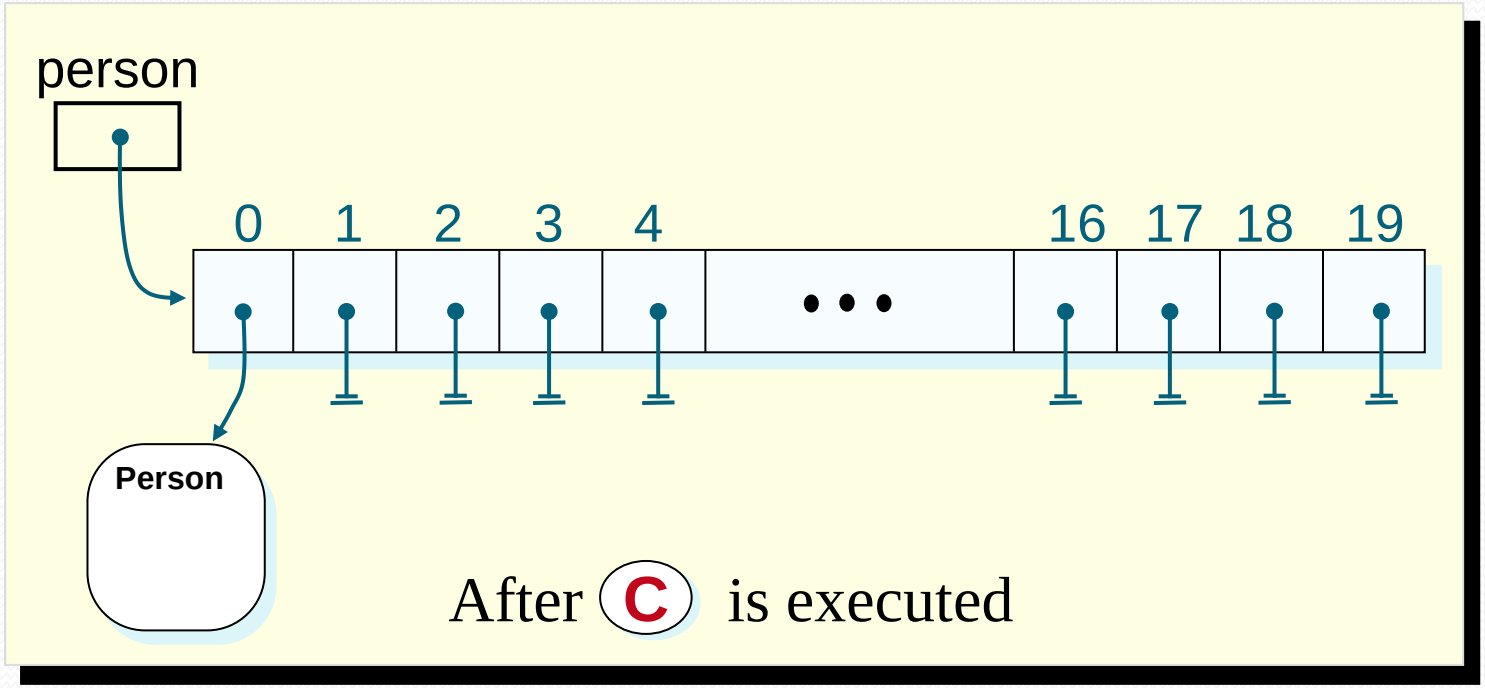
Code

C

```
Person[ ]  person;
person = new Person[20];
person[0] = new Person( );
```

One Person object is created and the reference to this object is placed in position 0.

State of Memory

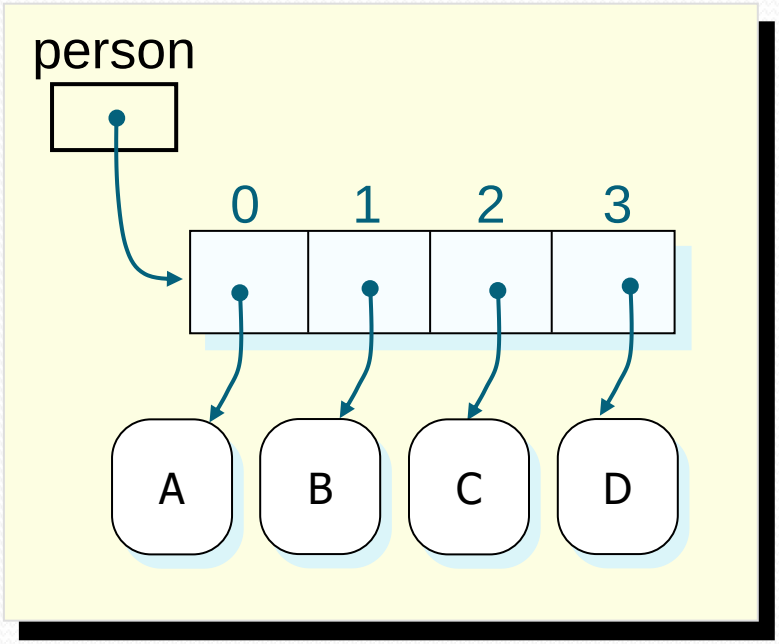


Object Deletion

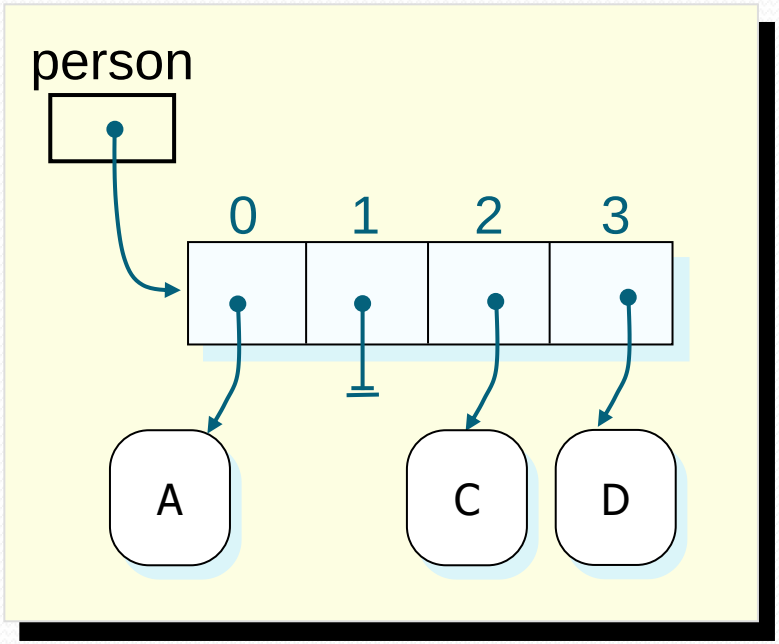
```
int delIdx = 1;
person[delIdx] = null;
```

A

Delete Person B by setting the reference in position 1 to null.



Before A is executed



After A is executed

The For-Each Loop

- This new for loop is available from Java 5.0
- The for-each loop simplifies the processing of elements in a collection
- Here we show examples of processing elements in an array

```
int sum = 0;

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

standard for loop

```
int sum = 0;

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

for-each loop

Processing an Array of Objects with For-Each

```
Person[] person = new Person[100];
//create person[0] to person[99]
```

```
for (int i = 0; i < person.length; i++) {
    System.out.println(person[i].getName());
}
```

standard for loop

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

for-each loop

For-Each: Key Points to Remember

- A for-each loop supports read access only. The elements cannot be changed.
- A single for-each loop allows access to a single array only, i.e., you cannot access multiple arrays with a single for-each loop.
- A for-each loop iterates over every element of a collection from the first to the last element. You cannot skip elements or iterate backward.

In-class Exercise

Create an array of persons, populate with some elements, and perform a search for a person with name “Bob”.

Sorting Strings

When you used Arrays.sort on an array of Strings, the JVM automatically uses the compareTo method to compare Strings and to put them in alphabetical order.

- Example:

```
public static void main(String[] args) {
    String[] names = {"Steve", "Joe", "Alice", "Tom"};
    //sorts the array in place
    Arrays.sort(names);
    System.out.println(Arrays.toString(names));
}
```

//output

```
[Alice, Joe, Steve, Tom]
```

Two-Dimensional Arrays

- Two-dimensional arrays are useful in representing tabular information.

Distance Table (in miles)

	Los Angeles	San Francisco	San Jose	San Diego	Monterey
Los Angeles	—	600	500	150	450
San Francisco	600	—	100	750	150
San Jose	500	100	—	650	50
San Diego	150	750	650	—	600
Monterey	450	150	50	600	—

Multiplication Table

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

Tuition Table

	Day Students	Boarding Students
Grades 1 – 6	$ 6,000.00	$ 18,000.00
Grades 7 – 8	$ 9,000.00	$ 21,000.00
Grades 9 – 12	$ 12,500.00	$ 24,500.00

Declaring and Creating a 2-D Array

Declaration

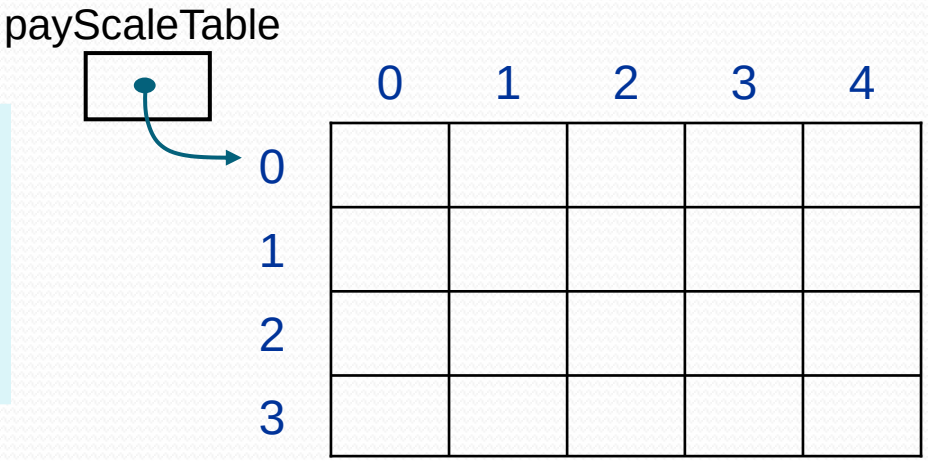
```
<data type>[][]  <variable>
```

Creation

```
<variable>  = new <data type> [ <size1> ][ <size2> ]
```

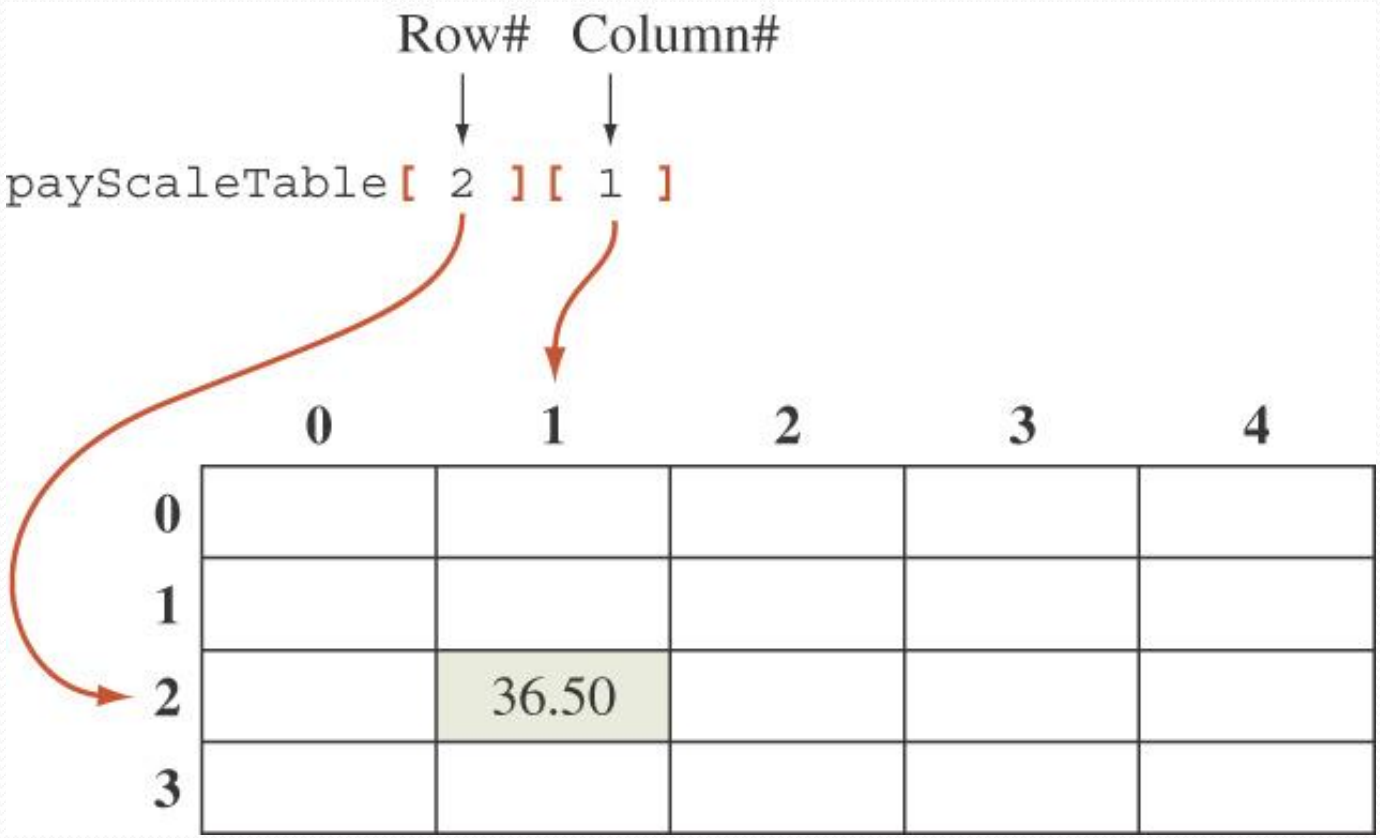
Example

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



Accessing an Element

- An element in a two-dimensional array is accessed by its row and column index.



Java Implementation of 2-D Arrays

- The sample array creation

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

is really a shorthand for

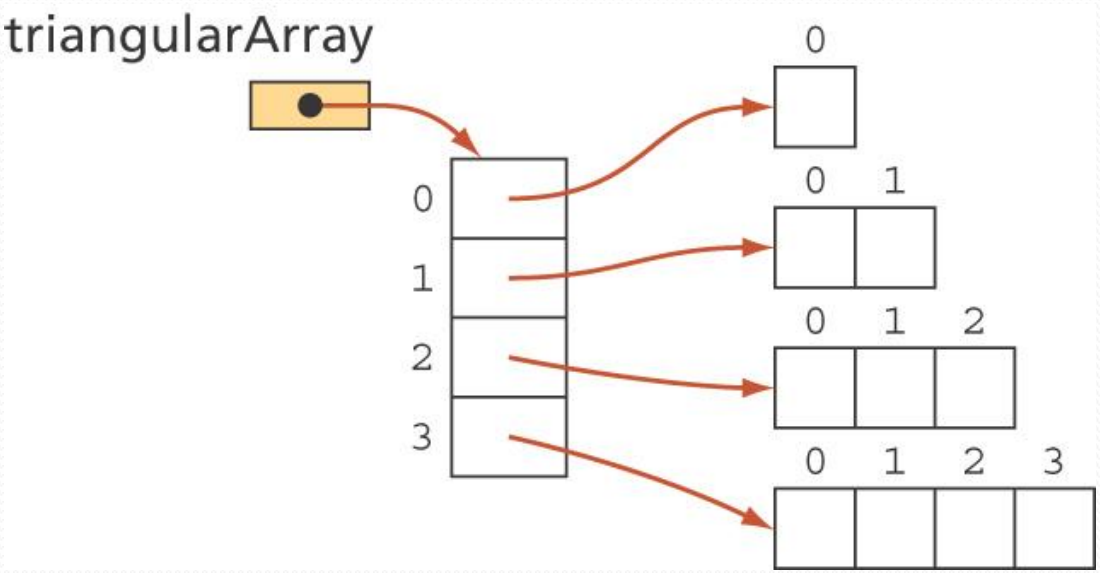
```
payScaleTable = new double [4][ ];

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

Two-Dimensional Arrays

- Subarrays may be different lengths.
- Executing

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

results in an array that looks like:



Sample 2-D Array Processing

- Find the average of each row.

```
double[ ] average = { 0.0, 0.0, 0.0, 0.0 };

for (int i = 0; i < payScaleTable.length; i++) {

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

        sum += payScaleTable[i][j];
    }

    average[i] = sum / payScaleTable[i].length;
}
```

Main Point

Arrays in Java support storage of multiple objects of the same type. Java supports multi-dimensional and ragged arrays; array copy and sort functions (accessible through the System and Arrays classes); and supports convenient forms of declaration and initialization. All CS data structures mirror the "existence" aspect of consciousness – the nervous system – whereas the *contents* of these structures mirrors the "intelligence" aspect; the pure potentiality of a data structure is as if brought to life by filling it with real data.

Date and Time API

- Before Java 8, `GregorianCalendar` and `Date` classes are used for representing date info.
- New in Java 8, `LocalDate` and `LocalDateTime` replaces them.
- LocalDate is *immutable;* operations that act on instances produce new instances – this is like Java's String class

LocalDate Sample Code

```
System.out.println("Today's date: " + LocalDate.now());
System.out.println("Specified date: " + LocalDate.of(2000, 1, 1));


//Formatting LocalDates as strings and reading date strings as
LocalDates
public static final String DATE_PATTERN = "MM/dd/yyyy";
public static LocalDate localDateForString(String date) {
        return LocalDate.parse(date,
        DateTimeFormatter.ofPattern(DATE_PATTERN));
}

public static String localDateAsString(LocalDate date) {
        return
        date.format(DateTimeFormatter.ofPattern(DATE_PATTERN));
}
```

See demo package: `lesson4.dates`