**Contents** **Topic 02 - Java Fundamentals**



## I. A Simple Java Program

FirstSample.java

```
public class FirstSample
{
   public static void main(String[] args)
   {
      System.out.println("Hello!");
   }
}
```

Access Modifier: **public**

- The keyword ***public*** is one of the **access modifiers**, that states how other parts can get the access.
  For example, here *public* is applied to *class FirstSample* meaning that outsiders can use the class.
- Except *public*, we also have other modifiers: *private*, *protected*, etc., We will cover them in next topic.

JAVA classes and .java files

- **Classes** are the building blocks for Java programs.
  By convention: start with an uppercase letter
- A .java file cannot have 2 or more public classes.
  The name of a public class has to be the same as the file name
- Case sensitive. FirstSample.java matches with class name FirstSample

The **main** method in Java

- **main()** does not return anything, thus **void**
- **main()** has to be inside a class; as a **static** method (ie. A method that does not operate on objects.).

**String[] args**

- Arguments can be supplied to **main()** as an array of strings. (Ref. Topic01)

**`print(..), println(..)`**

- General syntax to invoke a method: `object.method(parameters)`
- Use the `System.out` object and call its `println` method.
  - **`System.out.println("Hello!");`**
    prints "Hello!" + terminate the line (newline character '\n')
- Can be without argument: simply a blank line
  - **`System.out.println();`**
- `.print()` method
  - **`System.out.print("Hello!");`** ← not terminating the line (× "\n")

## II. Packages and the import statement

- **Packages** are groups of classes.
- The standard Java library is distributed over a number of packages
  e.g. **`java.lang, java.util, ..`**
- **`java.util`**
  - **`java.util`** contains the **`Scanner`** class (and many other classes)
  - We can use it like: **`java.util.Scanner`**
  - If we import **`java.util.*`**, then we can simply write: **`Scanner`**
- **The `import` statement:**
  - `import java.util.*;`
    We can use all classes in `java.util`
  - `import java.util.Scanner;`
    We can use Scanner

Main.java

```
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        System.out.print("Please enter the
        Scanner scannerObj = new Scanner(
        int y, m, d;
        y=scannerObj.nextInt();
```

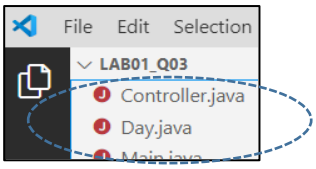
- **`java.lang` - Java fundamental classes**
  - No need to import `java.lang` (assumed already for all programs)
  - `java.lang` provides the `System` class , `Math` class, and many more..
- **Use of the `java.lang.System` class:**

```
public class FirstSample {

  public static void main(String[] args) {

    System.out.println("Hello!");

  }
}
```

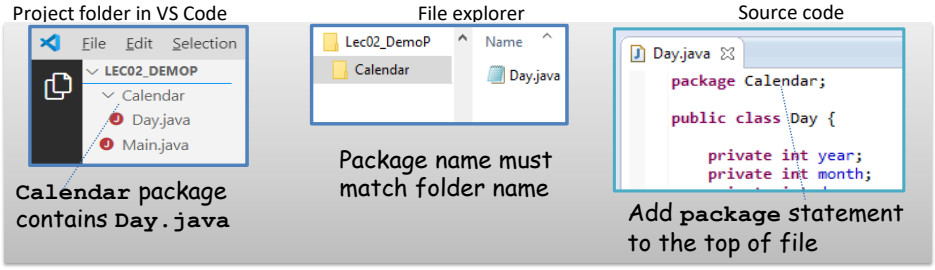
**Ref: JAVA API Documentation** https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html

## III. Creating Packages and "Default Package"

- We can group our source files into **packages**
  - A class not grouped into package is in "default package"




  - A class grouped into package is in the package folder and has the package statement:




To use the class, type `Calendar.Day` or add `import Calendar.*`

- Unluckily PASS doesn't compile packages at this moment.

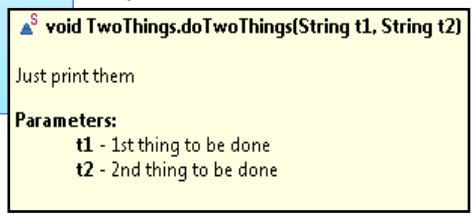
## IV. Comments

### Three ways of marking Comments

- Like **C++: //, /* . . . */**
- For **automatic documentation generation: /** . . . */**

```
/**
   Just print them
   @param t1 - 1st thing to be done
   @param t2 - 2nd thing to be done
 */
static void doTwoThings(String t1, String t2)
{
   System.out.println(t1);
   System.out.println(t2);
}
```

automatic documentation generation (by the javadoc program from JDK)




Some IDEs add * in front of every line, for visual style only.

```
/**
 * Just print them
 * @param t1 - 1st thing to be done
 * @param t2 - 2nd thing to be done
 */
```

# V. Data Types and Variables

## Overview of Data types

There are two kinds of types in the Java: **Primitive types** and **Reference types**

I. **Primitive types**: Java has 8 primitive types: `boolean, byte, short, int, long, float, double, char`

II. **Reference types** (~ pointers in C++) :

- The values of a reference type are references to objects.
  An *object* is an *instance* of a class.
  Note: The word "object" is often used interchangeably with "instance".
- Examples of built-in Java classes: `String, Math, Scanner`
  Examples of user-defined classes (Lab01): `Main, Day, Model, View, Controller`

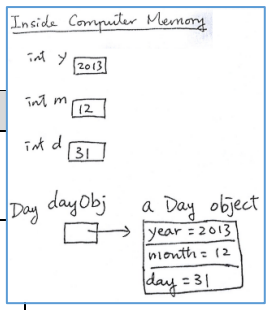
  We can create objects of these classes, using the **new** operator. After creation, we get the **object reference**.
  We often use a variable to hold the object reference;
  Then we can use the variable to access the object.

  e.g. Day dayObj = new Day(2013, 12, 31);
  System.out.println(dayObj.toString());
- An **array** is also a special kind of object

## Variables

We have seen that there are 2 types of data: Primitive and Reference types.

These data can be stored in variables: primitive values and reference values:




| Types of variables in JAVA | Information stored | Examples |
|---|---|---|
| **(i) Variables of Primitive Types** | The variables hold the exact values | `int x;`<br>`x = 689;` |
| **(ii) Variables of Reference Types** | The variables hold the references to **objects** (Like pointers in C++) | `Day d;`<br>`d = new Day(2016,1,20);`#1 |

#1: Use of a class: We use a class name as variable type, and use the class to create ("new") an object of its kind.

## Data Types – Integers

- Integer types

| | | |
|---|---|---|
| int | 4 bytes | –2,147,483,648 to 2,147,483, 647 (just over 2 billion) |
| short | 2 bytes | –32,768 to 32,767 |
| long | 8 bytes | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| byte | 1 byte | –128 to 127 |

- Java has no unsigned types
- Long integer numbers have a suffix L

```
//suffix 'L' is required for 2,223,123,123
System.out.println("Testing: " + 2223123123L); //Testing: 2223123123
long x;
x = 2223123123L;
System.out.println("x is: " + x); //x is: 2223123123
```

- To provide numbers in Binary, we need prefix: 0b
- To provide numbers in Hexadecimal, we need prefix: 0x

```
int x;
x = 0b11111111;
System.out.print(x); //shows 255

x = 0xFF;
System.out.print(x); //shows 255
```

## Data Types – Floating Point Types

- Floating Point Types
  - For numbers with fractional parts, (ie. not whole numbers, e.g. 1.1), and
  - For very large numbers (e.g. 5 x $10^{23}$ )
- The precision is limited (ie. keep a few significant digits).
- 2 Types:

| | | |
|---|---|---|
| float | 4 bytes | Approximately ±3.40282347E+38F (6–7 significant decimal digits) |
| double | 8 bytes | Approximately ±1.79769313486231570E+308 (15 significant decimal digits) |

- Suffix:
  - float - F
  - double – D (optional)

```
float x1 = 0.98765987659876598765F;
System.out.println(x1); //Shows 0.9876599

double x2=0.987659876598765987765D; //'D' is optional
System.out.println(x2); //Shows 0.987659876598766

double x3=0.98765987659876598765; //'D' is optional
System.out.println(x3); //Shows 0.987659876598766

double x4=9876598765987659876598765D; //'D' is optional
System.out.println(x4); //Shows 9.87659876598766E19
```

- Roundoff errors

```
System.out.println(2.0 - 1.1);  //0.8999999999999999
```

Reason of *Roundoff errors*:
computer uses binary number system; and there is no precise binary representation of a lot of fractions (e.g. 1/10)

Solution: use the `BigDecimal` class (Example is in the given code of this topic)

## Data Types – char and Escape Sequence

- Primitive type for characters: `char`
- Escape Sequence for special char values:

| Escape sequence | Name | Unicode value |
|---|---|---|
| \t | Tab | \u0009 |
| \n | Linefeed | \u000a |
| \" | Double quote | \u0022 |
| \' | Single quote | \u0027 |
| \\ | Backslash | \u005c |

## Data Types – Boolean

- `Boolean` type: `true, false`
  - We cannot convert between integers and Boolean values

```
boolean b=true;
int i = 3;
b = i; //Error!! Type mismatch - cannot convert from int to boolean
i = b; //Error!! Type mismatch - cannot convert from boolean to int
```

## Declaration of Variables

- Every variable has a type:
  - `double salary;`
  - `int vacationDays;`
  - `long earthPopulation;`
  - `boolean done;`

- Common ways to name: start with lowercase letter
  - `Box box; //Box is a class type and box is the variable.`
  - `Box aBox; //using "a" as prefix`
  - `Box bxJewels, bxCoins;`

- Must explicit initialize before use

```
int x;
System.out.println(x); // ERROR--variable not initialized
```

```
int x = 12;
System.out.println(x);  //12
```

```
int x;
x= 28;
System.out.println(x);  //28
```

## VI. Constants

- Constants: Java keyword is `final`
- **`final`:** the value is set once and for all.
- Common ways to name a constant: all in uppercase
- Often given as **Method constants** or **Class constants**

```
public class MyApp
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      ..
   }
}
```

```
public class MyApp
{
    public static final double CM_PER_INCH = 2.54;
    public static void main(String[] args)
    {
       ..
    }
}
```

## VII. Operators: Arithmetic Operators, Relational Operators, Boolean (logical) Operators, Bitwise Operators

### (I) Arithmetic Operators: +, -, *, /, %, +=, -=, *=, /=, %=

**x/y**

- If both x and y are integers, denotes **integer division**, e.g. `3/4` is `0`
- Otherwise **floating-point division**, eg. `3.0/4` is `0.75`

**x/0**

- If x is integer => **division by zero exception** (run-time error)
- Otherwise, ie. x is floating point => `NaN` Not a number or `Infinity`

```
System.out.println(0/0);    → java.lang.ArithmeticException: / by zero
System.out.println(0%0);    → java.lang.ArithmeticException: / by zero
System.out.println(12.0/0); → gives Infinity
System.out.println(0.0/0);  → gives NaN
System.out.println(12.0%0); → gives NaN
System.out.println(0.0%0);  → gives NaN
```

### (II) Relational Operators: ==, !=, >, <, >=, <=

### (III) Boolean (logical) operators: &&[AND], ||[OR]

- Evaluated in "Short circuit" fashion
  I.e., The second argument is not evaluated if first argument already determines the value.

Example 1:

```
if ((isMember==true) || (calculateAge(..)>=65))
    System.out.print("Gift");
```

if `isMember` is true, then `calculateAge` doesn't need to (will not) run.

Example 2:

```
if (totCourses>0 && totalMarks/totCourses >=90)
    System.out.print("Well done!!");
```

if `totCourse` is `0`, then `totalMark/totCourses` will not be calculated (avoid run-time error "Division-by-zero")

### (IV) Bitwise Operators:

`& ("AND")` `| ("OR")` `^ ("XOR")` `~ ("NOT")` `<< (left-shift)` `>> (right-shift)`

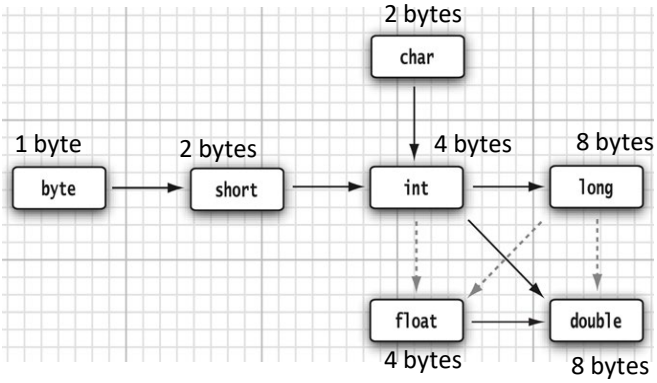
```
Example:
  int n1, n2, n3;
  n1 = 0xFE; (ie. 0b11111110)
  n2 = n1 ^ 0xFF; //set n2 to 0b00000001 (0b11111110 XOR 0b11111111)
  n3 = n2 << 4; //set n3 to 0b00010000 (left-shift 0b00000001 by 4 bits)
```

## VIII. Conversion between numbers, type casting

**Legal conversions:**

Case 1: Without information loss, or

Case 2: Just to lose precision




```
double d; float f; int i=2147483647;
d=i;
f=i;
System.out.println(f); //output: 2.14748365E9
System.out.println(d); //output: 2.147483647E9
System.out.println(i); //output: 2147483647
```

Case 3: Though legal, but may lose information. For these cases explicit **type casting** is needed:

```
int i=97; char c;
c=(char)i;
System.out.println(i); //output: 97
System.out.println(c); //output: a
```

Q: Why `int->char` may cause information lost?
A: `int` is 4 bytes, can hold big range of values
But `char` is 2 bytes only

## IX. Parentheses and Operator Hierarchy

When one expression contains 2 or more operators, then

The order of Evaluation depends on precedence level and associativity.

E.g.

```
int x = 12345 / (4 + 5 * 7 - 2);
              ④   ②   ①   ③
```

- The precedence level of / and * are higher than the precedence level of + and -
- To override the above ordering, we add () for grouping.

the precedence level of () is high

precedence level

| Operators | Associativity |
|---|---|
| () | Left to right |
| * / % | Left to right |
| + - | Left to right |
| < <= > >= | Left to right |
| == != | Left to right |
| && | Left to right |
| \|\| | Left to right |
| = += -= *= /= %= | Right to left |

**Exercise:**

**Q1**. For each underlined expression below, mark the steps with ①, ②:

(i). `System.out.println(1234 / 100 / 10) ;`

(ii) `System.out.println(1234 * 60 % 24);` (note: * and % have the same precedence)

(iii). `int a,b; a = b = 10;`

**Q2**. Delete the wrong items below (*):

In * (i) / (ii) / (iii), we say that the associativity is left-to-right.

In * (i) / (ii) / (iii), we say that the associativity is right-to-left.

Note:
When operators have the same **precedence level**, then **the associativity rules** of the operators decide the order of evaluation.

# X. Strings & StringBuilder

**Strings (`java.lang.String`)**

- A `String` object contains a sequence of Unicode characters (code units in UTF-16 encoding)
- String variables are references to string objects

```
String s = new String("Hello");
```
or
```
String s = "Hello";
```
← Shorthand

- **`.length`** method yields number of characters
- `""` is the empty string of length 0, different from `null`

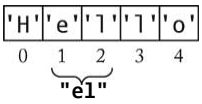
```
if (s!= null && s.length() != 0)
```
← Check for non-null and non-empty

- **`.charAt`** method yields `char`:

```
char c = s.charAt(i);
```

- **`.substring`** method yields substrings:

```
String greeting = "Hello";
String s = greeting.substring(1,3);
```




It means from position 1 inclusive to position 3 **exclusive**

- `+`

```
String greeting = "Hello"; String s;
s = 1000 + " " + greeting; // "1000 Hello"
s = 1000 + ' ' + greeting; // _____________
```

- Use **`.equals`** to compare **string contents**:

```
String greeting = "Hello";
String part = greeting.substring(1, 3);
if (part == "el") {..} //NO!
if (part.equals("el")) {..} //OK
```
← == tests whether both refers to the same object (not compare contents)

- Converting Strings to Numbers: **Integer.parseInt**

```
String input = "7";
int n = Integer.parseInt(input); // n gets 7
```

- Strings are **immutable**:
  - No method can change a character in an existing string
  - To turn greeting from "Hello" to "Help!", it is not so convenient:

```
greeting = greeting.substring(0,3)+"p!";
```
← actually a new string object

For the original string object which was previously referred by greeting, Java has the *Garbage Collection* mechanism to recycle the unused memory.

**StringBuilder (java.lang.StringBuilder)**

```
StringBuilder sb = new StringBuilder();
sb.append("Hello ");
sb.append(name); //suppose name is "Peter"
String result=sb.toString(); //gives "Hello Peter"
```

- If more concatenation work is needed, using + for string concatenation is inefficient ☹ reason: it actually creates new string objects
- **`StringBuilder`** object – can manipulate characters within itself. ☺
- Other `StringBuilder` methods for handling character contents: **`setCharAt, insert, delete`**

## XI. Input (Console, File, Input from String)

### (1) Reading input from Console

- Construct `Scanner` from input stream (e.g. `System.in`)
  `Scanner in = new Scanner(System.in);`
- `.nextInt, .nextDouble` reads next `int` or `double`
  `int n = in.nextInt();`
- `.next` reads next string (delimited by whitespace: space, tab, newline; discard leading whitespace)
- `.nextline` reads until `newline` and removes `newline` from the stream.
  Sometimes we need `.nextline` to remove extra line break (Learn from Lab03)
- `.close` closes the stream

```
Scanner in = new Scanner(System.in);
String s1,s2;
s1 = in.next(); //type "   Today    is a good day."
s2 = in.nextLine();
System.out.println(s1); //"Today"
System.out.println(s2); //"    is a good day“
in.close();
```

```
   Today    is a good day.
Today
    is a good day.
```

Rundown

### (2) Reading input from a file

- Construct `Scanner` from a `File` object
  - `Scanner inFile = new Scanner(new File("c:\\data\\case1.txt"));`
- `.hasNext()` checks whether there is still “next string” in the file.

```
Scanner inFile = new Scanner(new File(fileName));

while (inFile.hasNext()) {
       String line = inFile.nextLine();
       ..
}
inFile.close();
```

### (3) Reading input from another string

```
Scanner inData = new Scanner(str); //where str is a String
//.. apply .hasNext(), .next(), .close() etc..
```

Example: Read a of words and show them line by line:

```
System.out.print("Enter a line of words: ");

Scanner scannerConsole = new Scanner(System.in);
String str = scannerConsole.nextLine();

Scanner scannerStr = new Scanner(str);
while (scannerStr.hasNext())
       System.out.println(scannerStr.next());

scannerStr.close();
scannerConsole.close();
```

Output

```
Enter a line of words: Have a good day!
Have
a
good
day!
```

# XII. Output (System.out.printf, String.format, PrinterWriter)

### Formatted Output – using System.out.printf()

- Using **.print, .println** for floating-point values (problem):

```
double x = 10000.0 / 3.0;
System.out.println(x); //prints 3333.3333333333335
```

- Using **.printf** – formatted output (solution)

```
double x = 10000.0 / 3.0;
System.out.printf("%8.2f", x);//prints  3333.33
```

Field width of 8 characters
Precision of 2 characters
=> Result has a leading space and 7 characters

- Using **.printf** – multiple parameters

```
System.out.printf("Hi %s. Next year you'll be %d\n", name, (age+1));
```

- Conversion characters (%f, %d, %s):

| Conversion character | Type | Example |
|---|---|---|
| d | Decimal integer | 159 |
| x | Hexadecimal integer | 9f |
| o | Octal integer | 237 |
| f | Fixed-point floating-point | 15.9 |
| e | Exponential floating-point | 1.59e+01 |
| s | String | Hello |
| c | Character | H |
| b | boolean | true |
| % | The percent symbol | % |

- Similar method to create a string

```
String msg = String.format("Hi %s. Next year you'll be %d", name, age+1);
```

- Output to a file (Create a PrinterWriter object, then apply .print etc..)

```
PrintWriter out = new PrintWriter("c:\\report\\myfile.txt");
out.println("My GPA is 4.0");
out.close();
```

## XIII. Control flow

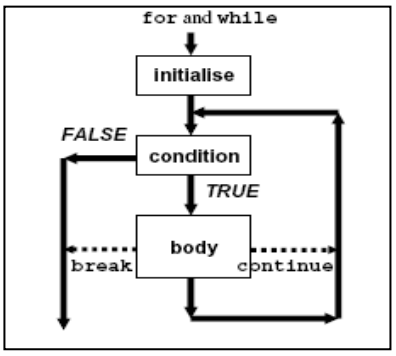
- Control structures (similar to C++):
  **if if..else switch-case while do-while for**
- Block Scope (compound statement inside {})
- Cannot declare identically named variables in 2 nested blocks:

```
public static void main(String[] args)
{
   int n;
   . . .
   {
     int k;
     int n; // ERROR--can't redefine n in inner block
     . . .
   }
}
```

- Declaring a variable in a for-loop:

```
for (int i = 1; i <= 10; i++)
{
  . . .
}
// i no longer defined here

for (int i = 11; i <= 20; i++) // OK to define another variable named i
{
  . . .
}
```

- Using break and continue:
  **break**
  means "immediately halt execution of this loop"
  **Continue**
  means "skip to next iteration of this loop."




We should use them only if that improves coding quality !

```
// Find x in an array A

bFound=false;
for (i=0;i<n;i++)
{
    if (A[i]==x)
    {
        bFound=true;
        break;
    }
}
```

Or

```
// No need to use break
bFound=false;
for (i=0;i<n && !bFound;i++)
{
      if (A[i]==x)
            bFound=true;
}
```

```
// read in 10 numbers
// and handle only the positive ones
for (i=0; i<10; i++)
{
    x=scannerObj.nextInt();
    if (x<0)
    {
        System.out.println("Wrong");
        continue;
    }

    .. // processing of x
}
```

# XIV. Arrays

- An array is a collection of elements **of the same type**
- Index is zero-based

```
int[] arr; // int[] is the array type; arr is the array name
           // int arr[]; is also okay, but not welcome by Java fans

arr = new int[5]; //create the array;
arr[0] = 3;
arr[1] = 25;

for (int i=0;i<arr.length;i++) //use .length to tell the array size
    System.out.println(arr[i]);
```

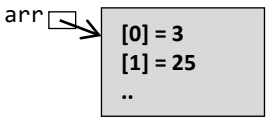
Output

```
3
25
0
0
0
```

Initialized values
For number elements : 0
For boolean elements: false;
For object elements: null

`Once created, cannot change size`
If extension is needed, make `arr` to refer to a new larger array and copy the old contents.

- Array variable is a reference




- Styles of array declaration:

(1)
```
int[] arr;
arr = new int[5];
arr[0] = 3;
arr[1] = 25;
```

(2)
```
int[] arr = new int[5];
arr[0] = 3;
arr[1] = 25;
```

(3)
```
int[] arr = {3,5,0,0,0};
```

Shorthand - Declaration with initializers

- Reinitializing **an array variable**

```
int[] arr = {3,5,0,0,0};
arr = new int[] {1,2,3,4,5,6,7,8};
```

← a new array

For the original array which was previously referred by arr, Java has a *Garbage Collection* mechanism to recycle the unused memory.

- **The "for each" loop:**
  - Syntax: for (variable : collection) statement
  - Example:

```
for (int x: arr)
    System.out.println(x);
```

← "for each" loop - goes through each element as **x**

Practice: Use more "for-each" loop from now on ! ☺

- **Arrays.toString**
  - Provided by the `java.util.Arrays` class
  - Returns a string representation of the array

```
int[] arr = {3,5,0,0,0};
System.out.println(Arrays.toString(arr));
```

Output

```
[3, 5, 0, 0, 0]
```

- Sorting with **Arrays.sort**

```
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

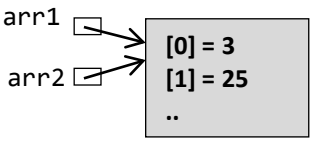
Output

```
[0, 0, 0, 3, 5]
```

- **Array copying**

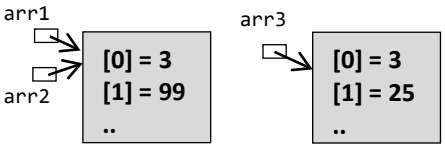
(1) Copying reference – not really creating new array

```
arr1 = new int[] {3,25,0,0,0};
arr2 = arr1;
```




(2) Copying as a new array: `Arrays.copyOf`
Syntax: `Arrays.copyOf(originalArray, newSize);`

```
int[] arr1, arr2, arr3, arr4, arr5;
arr1 = new int[] {3,25,0,0,0}; // 5 elements

arr2 = arr1;
arr3 = Arrays.copyOf(arr1, 4); // only want 4 elements

arr1[1]=99;

System.out.println(Arrays.toString(arr1));
System.out.println(Arrays.toString(arr2));
System.out.println(Arrays.toString(arr3));
```




Output

```
[3, 99, 0, 0, 0]
[3, 99, 0, 0, 0]
[3, 25, 0, 0]
```

`Arrays.copyOfRange` is another useful method. Learn it in Lab03.

- **Multidimensional array**

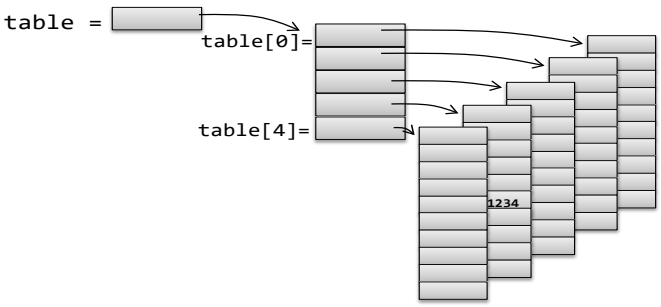
2D array:
- Is a 1D array of some 1D arrays

5 rows, 10 columns

```
int[][] table = new int[5][10];
table[3][5]=1234; // set 4th row, 6th column to 1234
for (int[] arr1D: table)
   System.out.println(Arrays.toString(arr1D));
```

Output

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1234, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```




Each element in the 2D array is a 1D array !

Ragged Array:
- Different rows have different lengths
  It is easy to do so in Java.

```
String[][] helpers = {
        {"Helena", "Kit", "Jason"},
        {"Helena", "Kit", "Jason"},
        {"Kit", "Jason"},
        {"Helena", "Kit"},
        {"Helena"}
};

System.out.println("Helpers for T01-T05:");
System.out.println("====================");

for (String[] arr1D: helpers)
        System.out.println(Arrays.toString(arr1D));
```

Output

```
Helpers for T01-T05:
====================
[Helena, Kit, Jason]
[Helena, Kit, Jason]
[Kit, Jason]
[Helena, Kit]
[Helena]
```