
School of Computing

# CS1020 Data Structures and Algorithms I
# Lecture Note #1

## Introduction to Java

# Lecture Note #1: **Intro to Java**

- **Objectives:**
  - Able to start writing Java programs
  - Able to translate most C programs learned in CS1010 into Java programs

- **Reference:**
  - Chapter 1
    - Section 1.1 (excludes Arrays) to Section 1.3: pages 27 to 45
    - Section 1.7 (excludes Console class): pages 73 to 77

# Outline

1. Brief history and background
2. Run cycle
3. Basic program structure
4. Basic Java elements
   - 4.1 Arithmetic Expressions
   - 4.2 Control Flow Statements and Logical Expressions
   - 4.3 Basic Input and Output
   - 4.4 Function

# 1. Java: Brief History & Background

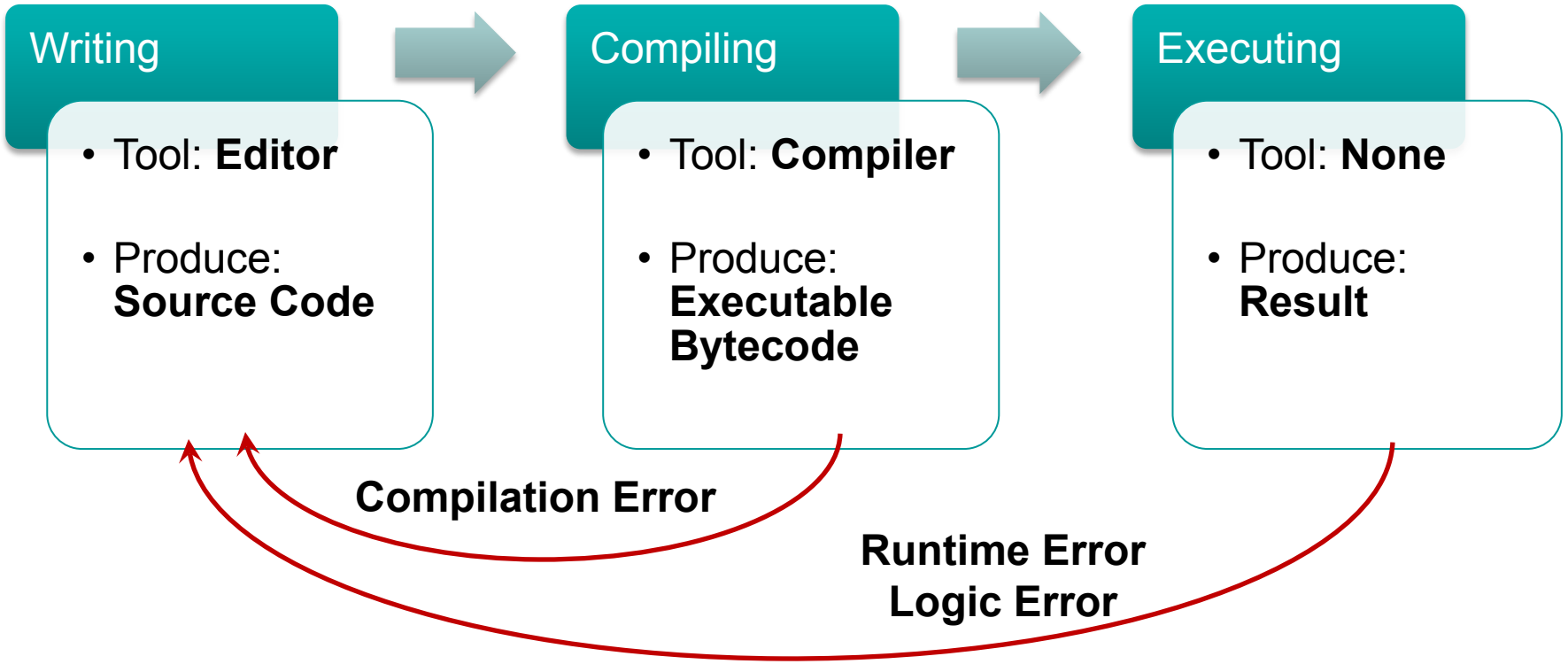
- Developed by **James Gosling:**
  - in **1995**
  - at **Sun Microsystems** (acquired by **Oracle** in 2010)

- Use **C** and **C++** as the foundation
  - "Cleaner" in syntax
  - Less low-level functionality
    - Shield the user from low-level machine interaction
  - More uniform object model

- Some selling points:
  - **Write Once, Run Everywhere™**
    - Compiled binary can be executed **across different platforms**
  - Extensive and well documented standard library

# 2. Run Cycle: Quick Recap

- **Run Cycle:**
  - The process of ***writing, compiling*** and ***executing*** a program



# 2. Run Cycle for Java Program

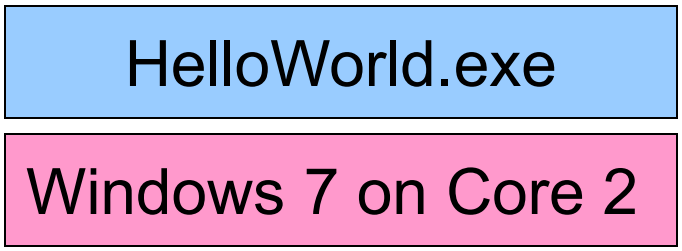
- **Writing / Editing Program**
  - Any text editor
  - Source code must have a `.java` extension
    - e.g. `Hello.java`
- **Compiling Program**
  - Use Java compiler `javac`
    - e.g. "`javac Hello.java`"
  - Compiled binary has `.class` extension:
    - e.g. `XXXX.class`
  - The binary is also known as **Java Executable Bytecode**
- **Executing Binary**
  - Run on a ***Java Virtual Machine (JVM)***
    - e.g. "`java XXXX`" (leave out the **.class** extension)
  - Note the difference here compared to normal C executable

# 2. Compile Once, Run Anywhere? How?

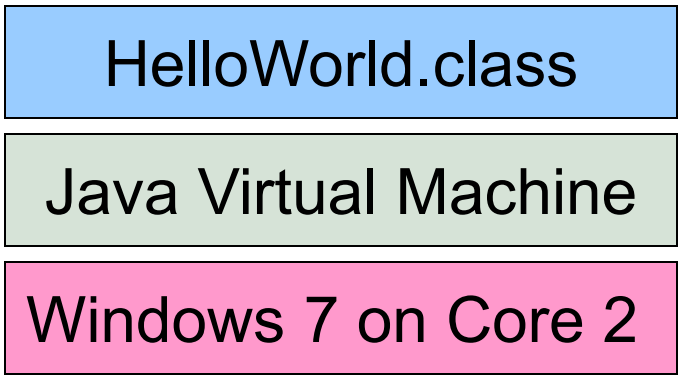
- Normal executable files are directly dependent on the OS / Hardware
  - Hence, an executable file is usually not executable on different platforms

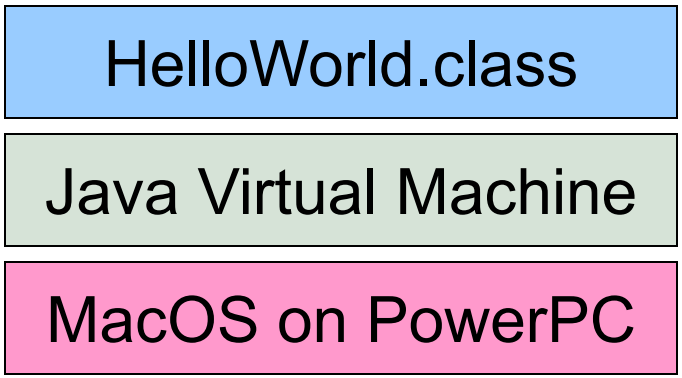
- Java overcomes this by running the executable on an **uniform hardware environment** simulated by software
  - The hardware environment is know as the **Java Virtual Machine (JVM)**
  - So, we only need a **specific JVM** for a particular platform to execute all Java Bytecodes without recompilation

# 2. Java Execution Illustration




Normal executable are tied to a specific platform ( OS + Hardware )




JVM provides a uniform environment for Java bytecode execution.

# 3. Java: Basic Language Introduction

- Cover the elementary language components:
  - Basic Program Structure
  - Primitive data types and simple variables
  - Control flow and repetition statements

- Purpose: ease you into the language
  - You can attempt to "translate" a few simple C programs into Java syntax to familiarize yourself

- Note:
  - Many important concepts will be introduced later
  - Do not take this section as the full language overview!

# 3. Hello World, Again!

```c
#include <stdio.h>

int main( )
{
    printf("Hello World!\n");

    return 0;
}
```

HelloWorld.c

**C program** to print out the message "Hello World!"

```java
import java.lang.*;     //optional

class HelloWorld {

  public static void main(String[] args) {

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

  }

}
```

HelloWorld.java

**Java program** to print out the message "Hello World!"

# 3. Key Observations (1/2)

- Library in Java is known as **package**
  - Packages are organized into hierarchical grouping
  - E.g., the "**`System.out.println()`**" is defined in the package "**`java.lang.system`**"
    - i.e. "**`lang`**" (language) is a group of packages under "**`java`**" (the main category) and "**`system`**" is a package under "**`lang`**"

- To use a predefined library, the appropriate package should be **imported**:
  - Using the "**`import XXXXXX`**;" statement
  - All packages under a group can be imported with a "**`*`**" wildcard

- Packages under "**`java.lang`**" are imported **by default**:
  - i.e. the import statement in this example is optional

# 3. Key Observations (2/2)

- The main method (function) is now enclosed in a "**class**"
  - The idea of class will be covered in lecture 2
  - There should be only one **main method** in a program, which serves as the execution starting point
  - Each source code file contains **one or more classes**
    - There are restrictions which will be covered later
  - Each class will be compiled into a separate **`XXXX.class`** bytecode
    - The "**`XXXX`**" is taken from the class name ("**`HelloWorld`**" in this example)

# 4.1 Arithmetic Expressions

# 4.1 Identifier and Variable

- **Identifier** is a **name** that we associate with program entity
- Java Identifier Rule:
  - Can consists of letters, digits, underscore (_) and **dollar sign ($)**
  - Cannot begin with a digit

- **Variable** is used to store data in a program
  - A variable must be declared with a specific data type

# 4.1 Numeric Data Types

- Summary of numeric data types in Java:

| | Type Name | Range |
|---|---|---|
| Integer Data Types | `byte` | $-2^7$ to $2^7-1$ |
| | `short` | $-2^{15}$ to $2^{15}-1$ |
| | `int` | **$-2^{31}$ to $2^{31}-1$** |
| | `long` | $-2^{63}$ to $2^{63}-1$ |
| Floating Point Data Types | `float` | Negative: -3.4028235E+38 to -1.4E-45<br>Positive: 1.4E-45 to 3.4028235E+38 |
| | `double` | **Negative: -1.7976931348623157E+308 to -4.9E-324**<br>**Positive: 4.9E-324 to 1.7976931348623157E+308** |

- You are strongly encouraged to use:
  - `int` for integers
  - `double` for floating point numbers

# 4.1 Numeric Operators

Higher Precedence

| | | |
|---|---|---|
| `()` | Parentheses Grouping | Left-to-right |
| `++, --` | Postfix incrementor/decrementor | Right-to-left |
| `++, --`<br>`+, -` | Prefix incrementor/decrementor<br>Unary +, - | Right-to-left |
| `%` | Remainder of integer division | Left-to-right |
| `*, /` | Multiplication, Division | Left-to-right |
| `+, -` | Addition, Subtraction | Left-to-right |
| `=`<br>`+= -= *= /= %=` | Assignment Operator<br>Shorthand Operators | Right-to-left |

- Evaluation of numeric expression:
  - Determine grouping using precedence
  - Use associativity to differentiate operators of same precedence
  - Data type conversion is performed for operands with different data type

# 4.1 Numeric Data Type Conversion

- When operands of an operation have differing types:
  1. If one of the operands is `double`, convert the other to `double`
  2. Otherwise, if one of them is `float`, convert the other to `float`
  3. Otherwise, if one of them is `long`, convert the other to `long`
  4. Otherwise, convert both into `int`

- When value is assigned to a variable of differing types:
  - **Widening (Promotion):**
    - Value has a smaller range compared to the variable
    - Converted automatically
  - **Narrowing (Demotion):**
    - Value has a **larger range** compared to the variable
    - **Explicit type casting is needed**

# 4.1 Data Type Conversion

- Conversion mistake:

```
double d;
int i;

i = 31415;
d = i / 10000; //attempt to get 3.14
```

What's the mistake? How do you correct it?

- Type casting:

```
double d;
int i;

d = 3.14159;
i = (int) d;   //attempt to get 3
```

The "`(int) d`" expression is known as **type casting**

**Syntax:**

`(datatype) value`

**Effect:**
The `value` is converted explicitly to the data type stated if possible.

# 4.1 Problem: **Fahrenheit to Celsius**

- Write a simple Java program Temperature.Java to:
  - Convert a temperature reading in Fahrenheit to Celsius degree using the following formula:

```
celsius = (5 / 9) * (fahrenheit - 32);
```

  - Print out the result

- For the time being, you can hard code an example temperature in the program instead of reading it from user

# 4.2 Control Statements

Program Execution Flow

# 4.2 Selection Statements

```
if (a > b) {
    ...
} else {
    ...
}
```

- **`if-else`** statement
  - else-part is optional
- Valid condition:
  - **Must be a *boolean* expression**
  - **Unlike C, integer values are NOT valid**

---

```
switch (a) {
    case 1:
        ...
        break;
    case 2:
    case 3:
        ...
    default:
}
```

- **`switch-case`** statement
- Expression in **`switch()`** must evaluate to a value of **`char`**, **`byte`**, **`short`** or **`int`** type
- **`break`**: stop the fall-through execution
- **`default`**: catch all unmatched cases
  - Optional

# 5.2 Repetition Statements

```
while (a > b) {
    ... //body
}
```

- Valid conditions:
  - **Must be a boolean expression**
- **`while`** : check condition before executing body

```
do {
    ... //body
} while (a > b);
```

- **`do-while`**: execute body before condition checking

---

```
for (A; B; C) {
    ... //body
}
```

- **`A`** : initialization (e.g. `i = 0`)
- **`B`** : condition (e.g. `i < 10`)
- **`C`** : update (e.g. `i++`)
- Any of the above can be empty
- Execution order:
  - `A, B, body, C, B, body, C …`

# 4.2 Boolean Data Type [new in Java]

- Java provides an actual **boolean** data type
  - Store boolean value ***true*** or ***false***, which are keywords in Java
  - Boolean expression evaluates to either ***true*** or ***false***

**SYNTAX**

```
boolean variable;
```

**Example**

```
boolean isEven = false;
int input;

// code to read input from user omitted
if (input % 2 == 0)
      isEven = true;

if (isEven)
      System.out.println( "Input is even!" );
```

**Equivalent:**

```
isEven = ( input % 2 == 0 );
```

# 4.2 Boolean Operators

| | Operators | Description |
|---|---|---|
| Comparison Operators | < | lesser than |
| | > | larger than |
| | <= | lesser or equal |
| | >= | larger or equal |
| | == | equal |
| | != | not equal |
| Logical Operators | && | AND |
| | \|\| | OR |
| | ! | NOT |
| | ^ | EXCLUSIVE-OR |

Operands are variables / values that can be compared directly.

Examples:

```
X < Y
1 >= 4
```

Operands are boolean variable / expression.

Examples:

```
(X < Y) && (Y < Z)
(!isEven)
```

# 4.3 Basic Input / Output

Interacting with the outside world

# 4.3 Reading input: The **Scanner** Class

**PACKAGE**

```
import java.util.Scanner;
```

**SYNTAX**

```
//Declaration of Scanner "variable"
Scanner scVar = _initialization_;

//Functionality provided
scVar.nextInt();
```

Read an integer value from source

```
scVar.nextDouble();
```

Read a double value from source

```
......
```

Other data type, to be covered later

# 4.3 Reading Input: **Fahrenheit Ver 2.0**

```
import java.util.Scanner;

class TemperatureInteractive {

  public static void main(String[] args) {

    double fahrenheit, celcius;
    Scanner myScanner = new Scanner(System.in);

    System.out.print("Enter temperature in Fahrenheit: ");
    fahrenheit = myScanner.nextDouble();

    celcius = (5.0 / 9) * (fahrenheit - 32);
    System.out.println("Celcius: " + celcius);

  }

}
```

TemperatureInteractive.java

# 4.3 Reading Input: **Key Points (1/2)**

- The statement

```
Scanner myScanner = new Scanner(System.in);
```

  - Declare a variable "**`myScanner`**" of **`Scanner`** type
    - "**`myScanner`**" is just a variable name, i.e. you are free to rename it
  - The initialization "**`new Scanner(System.in)`**"
    - Construct a **`Scanner`** object
      - We will discuss more later
    - Attach it to the standard input "**`System.in`**" (which is the keyboard)
      - This scanner variable will receive input from this source
    - Scanner can attach to a variety of input source, this is just a typical usage

# 4.3 Reading Input: **Key Points (2/2)**

- After proper initialization, a Scanner object provides functionality to read value of various type from the input source

- The statement

```
fahrenheit = myScanner.nextDouble();
```

  - ***nextDouble()*** works like a function that returns a double value
  - The scanner object converts the input into the appropriate data type and returns it
    - in this case, user input from the keyboard is converted into double value

# 4.3 Writing Output: The Standard Output

- The **`System.out`** is the predefined output device
  - Refers to the monitor / screen of your computer

**SYNTAX**

```
//Functionality provided
System.out.print( output_string );

System.out.println( output_string );

System.out.printf( format_string, [items] );
```

```
System.out.print("ABC");
System.out.println("DEF");


System.out.printf("Very C-like %.2f\n", 3.14159);
```

# 4.3 Writing Output: `printf()`

- Java introduces **`printf()`** in Java 1.5
  - Very similar to the C version

- The format string contains normal characters and a number of specifier
  - Specifier starts with a percent sign (%)
  - Value of the appropriate type must be supplied for each specifier

- Common specifiers and modifiers:

| | |
|---|---|
| `%d` | for integer value |
| `%f` | for double floating point value |
| `%s` | for string |
| `%b` | for boolean value |
| `%c` | for character value |

**SYNTAX**

`%[-][W].[P]type`

`-`: For left alignment
`W`: For width
`P`: For precision

# 4.3 Problem: **Approximating PI**

- One way to calculate the PI π constant:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - .....$$

- Write ApproximatePi.java to:
  1. Ask the user for the number of terms to use for approximation
  2. Calculate the PI constant with the given number of terms
  3. Output the approximation in 6 decimal places

# 4.4 Function

Reusable and independent code unit

# 4.4 Function with a new name

- In Java, C-like function is known as **static/class method**
  - Denoted by the "**static**" keyword before return data type
  - Another type of method, known as **instance method** will be covered later

use without object
a static method CANNOT call a non-static method

```
class Factorial {

  public static int factorial (int n) {
       if (n == 0) return 1;
       return n * factorial(n-1);
  }

  public static void main(String[] args) {

     int n = 5;    //You can change it to interactive input

     System.out.printf("Factorial(%d) = %d\n", n, factorial(n));
  }
}
```

Factorial.java

# 4.4 Method Parameter Passing

- **All** parameters in Java are **passed by value**:
  - A copy of the actual argument is created upon method invocation
  - The method parameter and its corresponding actual parameter are two independent variable

- In order to let a method modify the actual argument:
  - An **object reference data type** is needed (similar to pointer in C)
  - Will be covered later

# 5 Array

A collection of homogeneous data

# 5.1 Array

- Array is the simplest way to store a collection of data of the same type (homogeneous)
- In Java, **array is an object**.

TestArray.java

```
class TestArray {

  public static void main(String[] args) {

    int[] arrayRef;

    arrayRef = new int[3];

    System.out.println(arrayRef.length);

    arrayRef[0] = 100;
    arrayRef[1] = arrayRef[0] - 38;
    arrayRef[2] = 88;
  }
}
```

**Sytnax:**
`datatype[] array_reference`

Construct the array

`length` is a **public attribute** of array reference type

After construction, array indexing works similarly to C

# 5.2 Array: **Simple Usage**

TestArrayUsage.java

```
class TestArrayUsage {

  public static void main(String[] args) {

    int[] arrayRef = { 100, 62, 88 };



    for (int element: arrayRef) {
      System.out.println(element);
    }



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

  }
}
```

Shortcut to construct the array and initialize the elements at the same time

**Syntax (Enhanced For-Loop):**
`for (datatype e: array_ref)`

Go through all elements in the array. "e" automatically refers to the array element sequentially in each iteration.

Equivalent version

# 5.3 Array: As a parameter

- As the reference to the array is passed into a method:
  - Any modification of the element in the method will affect the actual array

TestArraySwap.java

```
class TestArraySwap {
   public static void swap(int[] A, int i, int j) {
      int temp = A[i];
      A[i] = A[j];
      A[j] = temp;
   }

   public static void main(String[] args) {
      int[] arrayRef = { 100, 62, 88 };

      swap(arrayRef, 0, 2);

      for (int element: arrayRef) {
         System.out.println(element);
      }
   }
}
```

What is the output?

# 5.4 Array: As a return type

- Array can be returned from a method

TestArrayReturn.java

```
class TestArrayReturn {
   public static int[] makeArray(int size, int value) {
      int[] array = new int[size];

      for (int i = 0; i < array.length; i++)
         array[i] = value - i;
      return array;
   }

   public static void main(String[] args) {
      int[] arrayRef;

      arrayRef = makeArray(5, 99);

      for (int element: arrayRef) {
         System.out.println(element);
      }
   }
}
```

What is the output?

# 5.5 Array: **Common Mistakes** (1/3)

- length versus length()
  - To obtain length of a **`String`** object, we use the **`length()`** method
    - Example: **`str.length()`**
  - To obtain length (size) of an array, we use the **`length`** attribute
    - Example: **`arr.length`**
- Array index out of range
  - Beware of ArrayIndexOutOfBoundsException (exception is covered in section 3)

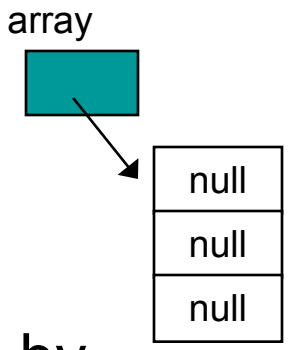
```
public static void main(String[] args) {
   int[] numbers = new int[10];
   . . .
   for (int i = 1; i <= numbers.length; i++)
      System.out.println(numbers[i]);
}
```

# 5.5 Array: **Common Mistakes** (2/3)

- When you have an array of objects, it's very common to forget to instantiate the array's objects.
- Programmers often instantiate the array itself and then think they're done – that leads to java.lang.NullPointerException

# 5.5 Array: **Common Mistakes** (3/3)

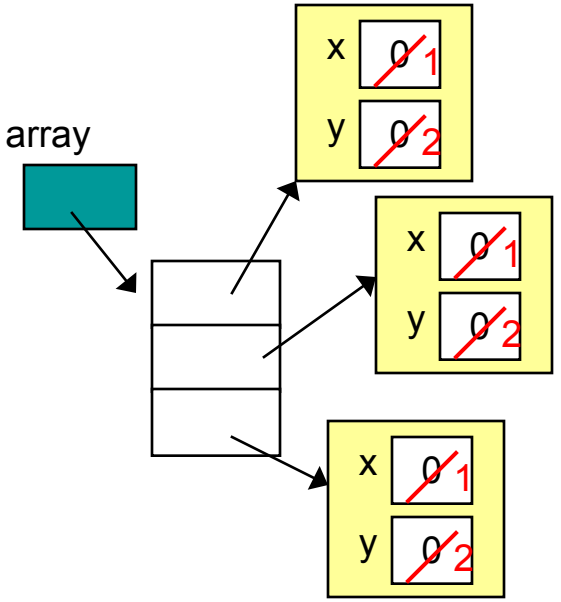
```
Point[] array = new Point[3];
for (int i=0; i<array.length; i++) {
   array[i].setLocation(1,2);
}
```




There are no objects referred to by array[0], array[1], and array[2]!

Corrected code:

```
Point[] array = new Point[3];
for (int i=0; i<array.length; i++) {
   array[i] = new Point();
   array[i].setLocation(1,2);
}
```

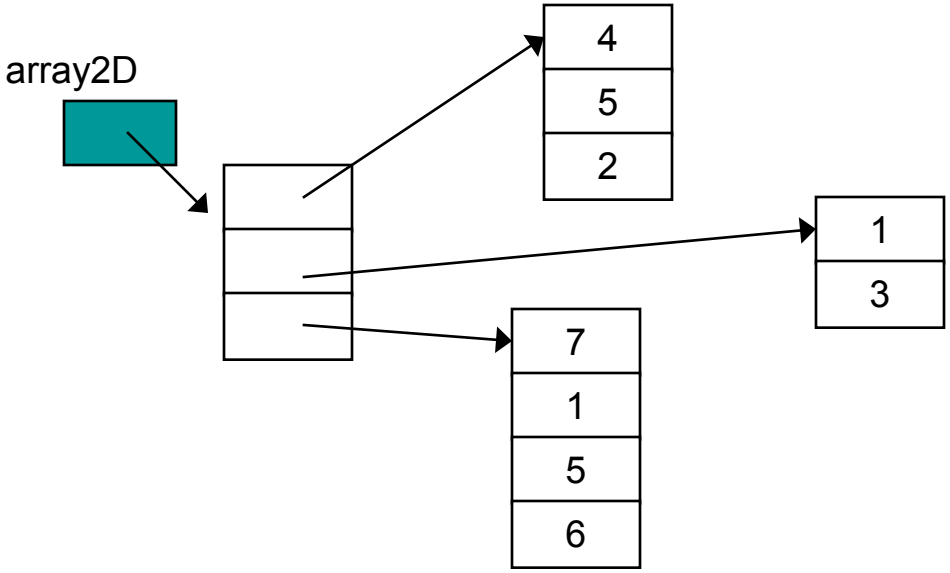


# 5.6 2D Array (1/2)

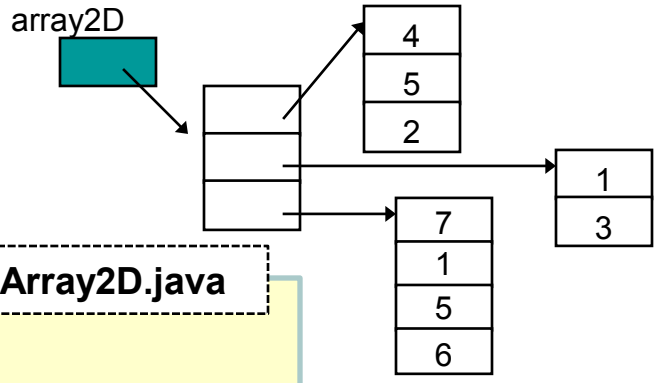
- A two-dimensional (2D) array is an array of array.
- This allows for rows of different lengths.

```
// an array of 12 arrays of int
int[][] products = new int[12][];
```

```
int[][] array2D = { {4,5,2}, {1,3},
                    {7,1,5,6} };
```



# 5.6 2D Array (2/2)




TestArray2D.java

```
class TestArray2D {

   public static void main(String[] args) {
      int[][] array2D = { {4, 5, 2}, {1, 3}, {7, 1, 5, 6} };

      System.out.println("array2D.length = " + array2D.length);
      for (int i = 0; i < array2D.length; i++)
         System.out.println("array2D[" + i + "].length = "
                            + array2D[i].length);

      for (int row = 0; row < array2D.length; row++) {
         for (int col = 0; col < array2D[row].length; col++)
           System.out.print(array2D[row][col] + " ");
         System.out.println();
      }
   }
}
```

# Summary

**Java Elements**

*Data Types:*
- *Numeric Data Types:*
  - *byte, short, int, float, double*
- *Boolean Data Type*

*Expressions:*
- *Arithmetic Expression*
- *Boolean Expression*

*Control Flow Statements:*
- *Selection Statements:*
  - *if-else, switch-case*
- *Repetition Statements:*
  - *while, do-while, for*

*Libraries:*
- *Simple Input/Output*

# Annoucement

- Lab 0 is in CodeCrunch and IVLE workbin
- Please use the 3 exercises to practice. 1% will be given when you pass all test cases.