OOP USING JAVA

Module 1:

The History and Evolution of Java

An overview of java

Data Types, Variables, and Arrays

Operators

THE HISTORY AND EVOLUTION OF JAVA

- Java and its applications
- Byte code
- Java Development Kit (JDK)
- Java Buzzwords
- Java Virtual Machine (JVM)
- The evolution of Java.

JAVA AND ITS APPLICATIONS

- Java is a general-purpose programming language that is class-based and object-oriented.
- It is structured in such a way that developers can write code anywhere and run it anywhere without worrying about the underlying computer architecture. It is also referred to as *write once, run anywhere* (WORA).
- Java is used for:
  - GUI applications
  - Web servers and applications servers
  - Middleware applications
  - Web applications
  - Mobile applications
  - Embedded systems
  - Enterprise applications

JAVA'S MAGIC: THE BYTECODE

- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- The original JVM was designed as an *interpreter for bytecode.*
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- Although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.
- JIT compiler, part of the JVM. Will compile selected portions of bytecode into executable code in real time, on a piece-by-piece, demand basis.

JAVA DEVELOPMENT KIT (JDK)

- The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets.
- It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.
- To run Java applications and applets, simply download the JRE.
- To develop Java applications and applets as well as run them, the JDK is needed.
- Java developers are initially presented with two JDK tools, java and javac. Both are run from the command prompt.
- The javac compiler is invoked to create .class files. Once the .class files are created, the 'java' command can be used to run the java program.

THE JAVA FEATURES(BUZZWORDS)

- **Simple**

  Easy to learn as it uses syntax similar to C/C++.

- **Secure**

- **Portable:** Write Once, Run Anywhere

- **Object-oriented:**

  “everything is an object”,primitive types, such as integers, are kept as high performance non objects.

THE JAVA FEATURES(BUZZWORDS)

- **Robust**
  - Memory Management :Garbage collector
  - Mishandled exceptions: Exception Handling
- **Multithreaded**
  - Parallel Computing for interactive systems
- **Architecture-neutral**
  - longevity(to withstand changes in operating systems) and portability
- **Interpreted and High performance**
- **Distributed**
  - *Remote Method Invocation (RMI). This feature enables a program to* invoke methods across a network
- **Dynamic**
  - run-time type information.

JAVA EVOLUTION

Version	Release date	End of Free Public Updates[5][6]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	**January 2019 for Oracle (commercial)** December 2020 for Oracle (personal use) At least September 2023 for AdoptOpenJDK	March 2025
Java SE 9	September 2017	March 2018	N/A
Java SE 10	March 2018	September 2018	N/A
Java SE 11 (LTS)	September 2018	**September 2023 for Oracle (commercial)** At least September 2022 for AdoptOpenJDK	September 2026
Java SE 12	March 2019	N/A for Oracle September 2019 for OpenJDK	N/A

Legend: Old version | Older version, still supported | **Latest version** | Future release

OBJECT ORIENTED PROGRAMMING

- **Two Paradigms**
  - Process oriented
  - Object oriented
- **Abstraction:** To hide the implementation.
  - Hierarchical abstractions.
- **The Three OOP Principles**
  - **Encapsulation:** is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
  - Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
  - a class is a logical construct; an object has physical reality.
  - Access specifiers are used to achieve encapsulation.
  - **Inheritance:** Inheritance is the process by which one object acquires the properties of another object.
  - it supports the concept of hierarchical classification.

OBJECT ORIENTED PROGRAMMING

- **Polymorphism**: *Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to* be used for a general class of actions.
- **Polymorphism, Encapsulation, and Inheritance Work Together:**

FIRST JAVA PROGRAM, EDIT-COMPILE-RUN CYCLE, JAVA ENVIRONMENT

- To compile or run java program, JDK need to be installed and configured with environment variable.
- A java program can be written on any text editor and the files should have java extension.
- For Compilation: javac filename.java
- For Executing: java classname

- Ref: https://youtu.be/vhBNV8no4CI

CONTROL STATEMENTS

- if
- if else
- Nested if
- else if ladder
- switch
  - break is optional in switch
  - Nested switch

ITERATION STATEMENTS

- while
- do while
- for
  - for as foreach.
  - for-each is essentially read only.
- Nested Loops

LEXICAL ISSUES

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- Whitespaces
  - Java is a free-form language, means that you do not need to follow any special indentation rules.
  - In Java, white space includes a space, tab, newline.
- Identifiers
  - Identifiers are used to name things, such as classes, variables, and methods.
  - Java is case-sensitive, so **VALUE is a** different identifier than **Value.**
  - Valid Identifiers:

AvgTemp	count	a4	$test	this_is_ok

  - Invalid Identifiers

2count	high-temp	Not/ok

LEXICAL ISSUES (CONTD..)

- Literals
  - A constant value in Java is created by using a *literal representation of it.*

100	98.6	'X'	"This is a test"

- *Comments*
  - *Single Line: //*
  - *Multiline: /* Comment */*
  - *Documentation: /** Statements */*
    - Allow you to embed information about your program into the program itself.
    - You can then use the Javadoc utility program (supplied with the JDK) to extract the information and put it into an HTML file.

LEXICAL ISSUES (CONTD..)

- **Separators**
  - The most commonly used separator in Java is the semicolon, it is often Used to terminate statements.

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference.
...	Ellipsis	Indicates a variable-arity parameter.
@	Ampersand	Begins an annotation.

LEXICAL ISSUES (CONTD..)

- **Keywords**

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Note: These keywords cannot be used as names for a variable, class, or method.

IDENTIFIERS IN JAVA

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- They must not begin with a number, lest they be confused with a numeric literal.
- Java is case-sensitive, so **VALUE is a different identifier than Value.**
- Valid Identifiers Ex

AvgTemp	count	a4	$test	this_is_ok

- Invalid Identifiers Ex

2count	high-temp	Not/ok

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

A CLOSER LOOK AT LITERALS

- Integer Literals
  - Any whole number is an integer Literal, Ex: 1,2,3,42 etc….
  - There are two other bases which can be used in integer literals, *octal (base eight) and hexadecimal (base 16).*
  - Octal values are preceded with 0. 0-7 is octal range.
  - Hexadecimal values preceded with 0x or 0X. The range of a hexadecimal digit is 0 to 15, so *A* through *F (or a through f) are substituted for 10 through 15.*
  - An integer literal can always be assigned to a long variable.
  - However, to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. This is done by appending an upper- or lowercase *L to the literal.*
  - *For example, 0x7fffffffffffffL* or 9223372036854775807L is the largest long.
  - An integer can also be assigned to a char as long as it is within range.

A CLOSER LOOK AT LITERALS(CONTD..)

- Floating Point Literals
  - Floating-point numbers represent decimal values with a fractional component.
  - can be expressed in either standard or scientific notation.
  - *Standard notation consists of a whole number* component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667.
  - *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.
  - The exponent is indicated by an *E or e followed by a* decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.
  - Floating-point literals in Java default to double precision.
  - To specify a float literal, you must append an *F or f to the constant.*
  - *You can also explicitly specify a double literal by* appending a *D or d.*
  - *The default double type consumes 64* bits of storage, while the less-accurate float type requires only 32 bits.

A CLOSER LOOK AT LITERALS(CONTD..)

- Boolean Literals
  - There are only two logical values that a boolean value can have, true and false.
  - The values of true and false do not convert into any numerical representation.
  - The true literal in Java does not equal 1, nor does the false literal equal 0.
  - In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

A CLOSER LOOK AT LITERALS(CONTD..)

- Charecter Literals
  - They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators.
  - A literal character is represented inside a pair of single quotes.
  - All of the visible ASCII characters can be directly entered inside the quotes, such as *'a', 'z', and '@'*.
  - For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as '\'' for the single-quote character itself and '\n' for the newline character.
  - There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, *'\141' is the letter 'a'*.
  - *For hexadecimal, you enter a backslash-u (\u), then* exactly four hexadecimal digits. For example, *'\u0061' is the ISO-Latin-1 'a' and '\ua432' is a Japanese Katakana character.*

A CLOSER LOOK AT LITERALS(CONTD..)

- **Character Escape Sequences**

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

A CLOSER LOOK AT LITERALS(CONTD..)

- **String Literal**
- String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

  “Hello World”

  “two\nlines”

  “\”This is in quotes\””“
- The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals.
- Strings must begin and end on the same line. There is no line-continuation escape sequence.

DATA TYPES, VARIABLES AND ARRAYS

- **Java Is a Strongly Typed Language**
- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches or errors that must be corrected, will be done before the compiler will finish compiling the class.

THE PRIMITIVE TYPES

- **Integers:** This group includes **byte, short, int, and long,** which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes **float and double**, which represent numbers with fractional precision.
- **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

THE PRIMITIVE TYPES

- **Byte**
  - The smallest integer type is **byte. This is a signed 8-bit type that has a range from –128 to 127.**
  - Especially useful when you're working with a stream of data from a network or file.
  - also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
  - Ex: byte b,c;

- **short**
  - short is a signed 16-bit type. It has a range from –32,768 to 32,767.
  - It is probably the least-used Java type.
  - Ex: short s,t;

THE PRIMITIVE TYPES

- **Int**
  - The most commonly used integer type is int.
  - It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.
  - variables of type int are commonly employed to control loops and to index arrays.
- **long**
  - long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.
  - The range of a long is quite large. This makes it useful when big, whole numbers are needed.

THE PRIMITIVE TYPES

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

THE PRIMITIVE TYPES

- **Float**
  - The type **float specifies a *single-precision value that uses 32 bits of storage.***
  - ***Single precision is*** faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small.
  - Variables of type **float are useful** when you need a fractional component, but don't require a large degree of precision.
  - Ex: float hightemp,lowtemp;
- **Double**
  - Double precision, as denoted by the **double keyword, uses 64 bits to store a value.**
  - **Double** precision is actually faster than single precision.
  - some modern processors that have been optimized for high-speed mathematical calculations.
  - All transcendental math functions, such as **sin( ), cos( ), and sqrt( ), return double values.**

THE PRIMITIVE TYPES

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

- **Char**
  - used to store characters is **char.**
  - Java **char is a 16-bit type.**
  - The range of a **char is 0 to 65,536.**
  - There are no negative **chars.**
  - The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

THE PRIMITIVE TYPES

- **Char**

```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

THE PRIMITIVE TYPES

- **Booleans**
  - It can have only one of two possible values, **true or false.**
  - This is the type returned by all relational operators, as in the case of **a < b. boolean is also the type *required by the conditional expressions that govern*** the control statements such as **if and for.**

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

VARIABLES

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- **Syntax:**
  - Data_type identifier [= value] ;

```
int a, b, c;                // declares three ints, a, b, and c.
int d = 3, e, f = 5;        // declares three more ints, initializing
                            // d and f.
byte z = 22;                // initializes z.
double pi = 3.14159;        // declares an approximation of pi.
char x = 'x';               // the variable x has the value 'x'.
```

VARIABLES

- **Dynamic Initialization**

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double a = 3.0, b = 4.0;

      // c is dynamically initialized
      double c = Math.sqrt(a * a + b * b);

      System.out.println("Hypotenuse is " + c);
    }
}
```

VARIABLES

- **Scope and Lifetime of Variables**
  - A block defines a *scope. Thus, each time you start a new block, you are creating a new scope.*
  - *A scope determines* what objects are visible to other parts of your program. It also determines the lifetime of those objects.

```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

VARIABLES

- variables are created when their scope is entered, and destroyed when their scope is left.

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

VARIABLES

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

OPERATORS

- **Arithmetic Operators**

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

- The operands of the arithmetic operators must be of a numeric type.
- You cannot use them on boolean types, but you can use them on char types, since the char type in Java is essentially, a subset of int.

OPERATORS(CONTD..)

- **The Bitwise Operators**
- Java defines several *bitwise operators that can be applied to the integer types, long, int, short,* char, and byte. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

OPERATORS(CONTD..)

- **Bitwise Logical Operators**

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

OPERATORS(CONTD..)

- **Relational Operators**

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

OPERATORS(CONTD..)

- **The ? Operator**
- Java includes *ternary (three-way) operator that can replace certain types of if-then-else* statements. This operator is the ?.
- The ? has this general form:

  expression1 ? expression2 : expression3

- Here, *expression1 can be any expression that evaluates to a boolean value. If expression1 is* true, then *expression2 is evaluated; otherwise, expression3 is evaluated.*
- *The result of the ?* operation is based on the expression1 evaluation.
- Both *expression2 and expression3 are required.*

TYPE CONVERSION AND CASTING

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- There are no automatic conversions from the numeric types to **char or boolean. Also, char and boolean are not compatible with each other.**
- **Implicit type conversions can be called as widening conversions.**
- **Explicit type conversion may result in losing the data, so it can be called as narrowing conversion.**
  - **Syntax: (target-type) value;**
- If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the byte's range).
- when a floating-point value is assigned to an integer type *truncation will occur.*

TYPE CONVERSION AND CASTING

- **Automatic Type Promotion in Expressions**
  - In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

  - The result of the intermediate term **a*b easily exceeds the range of either of its byte** operands.
  - Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
  - This means that the subexpression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

TYPE CONVERSION AND CASTING

- **The Type Promotion Rules**
  - All **byte, short, and char values are promoted to int.**
  - Then, if one operand is a long, the whole expression is promoted to long.
  - If one operand is a float, the entire expression is promoted to float.
  - If any of the operands is double, the result is double.

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

CONTROL STATEMENTS

- if
- if else
- Nested if
- else if ladder
- switch
  - break is optional in switch
  - Nested switch

ITERATION STATEMENTS

- while
- do while
- for
  - for as foreach.
  - for-each is essentially read only.
- Nested Loops

ARRAYS

- An *array is a group of like-typed variables that are referred to by a common name.*
- *Arrays of* any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.
- **One-Dimensional Arrays**
  - The general form of a one-dimensional array declaration is

    type var-name[ ];
  - Here, *type declares the base type of the array.*
  - *Ex:* int month_days[];
  - This declaration establishes the fact that month_days is an array variable,
  - No array actually exists.
  - To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days.

ARRAYS

- The general form of **new as it applies to one-dimensional arrays appears** as follows:
  - *array-var = new type[size];*
- to use **new to** allocate an array, you must specify the type and number of elements to allocate.
- The elements in the array allocated by **new will automatically be initialized to zero.**
- ***So,*** Arrays will be created in two steps, First array variable is created, then it will be physically created by using new keyword.

```
int month_days[];
Month_days=new int[12];
```

ARRAYS

- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.
- All array indexes start at zero.

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

ARRAYS

```
// An improved version of the previous program.
class AutoArray {
  public static void main(String args[]) {

    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                         30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

ARRAYS

```
// Average an array of values.
class Average {
  public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;

     for(i=0; i<5; i++)
       result = result + nums[i];

     System.out.println("Average is " + result / 5);
   }
 }
```

MULTIDIMENSIONAL ARRAYS

- **In Java, multidimensional arrays are actually arrays of arrays.**
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- **Ex: int twoD[][]=new int[4][5];**

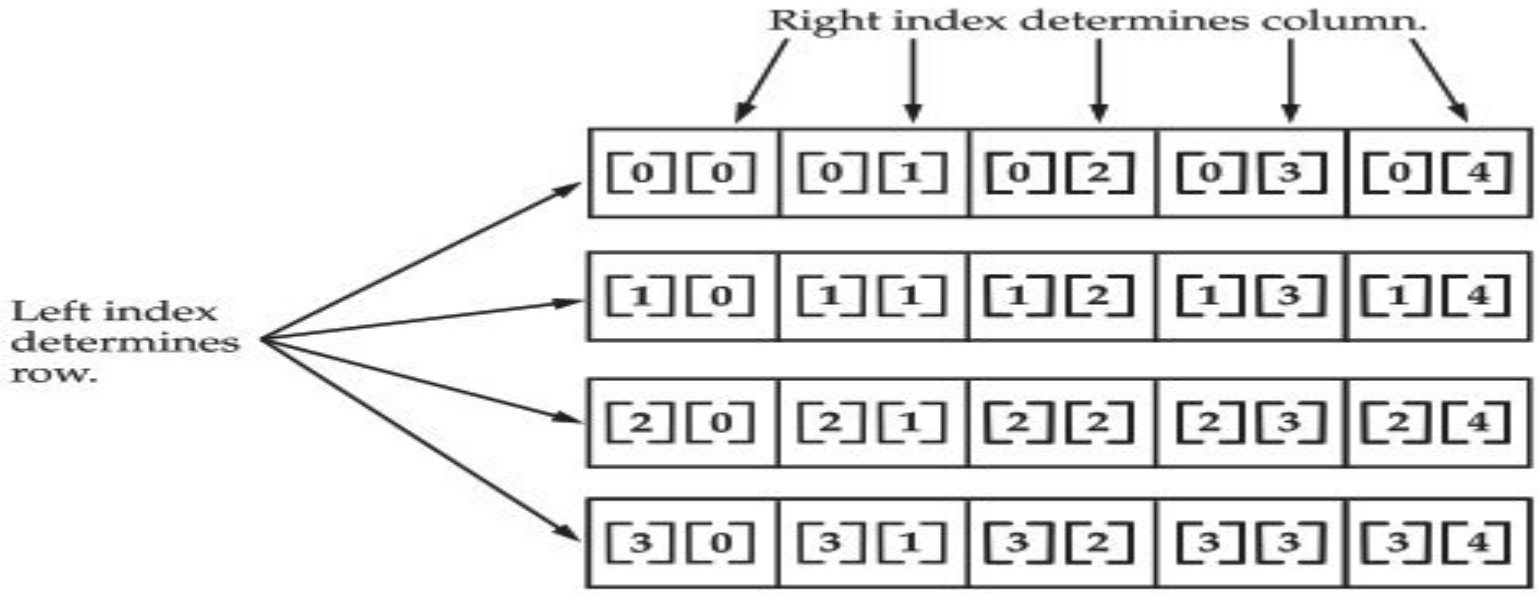

FIGURE 3-1 A conceptual view of a 4 by 5, two-dimensional array

MULTIDIMENSIONAL ARRAYS

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;

      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

MULTIDIMENSIONAL ARRAYS

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

MULTIDIMENSIONAL ARRAYS

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

MULTIDIMENSIONAL ARRAYS

```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

MULTIDIMENSIONAL ARRAYS

This program generates the following 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  1  2  3  4
0  2  4  6  8
0  3  6  9  12

0  0  0  0  0
0  2  4  6  8
0  4  8  12  16
0  6  12  18  24
```

ALTERNATIVE ARRAY DECLARATION SYNTAX

- There is a second form that may be used to declare an array is:
  - type[ ] var-name;
  - int al[] = new int[3];
  - int[] a2 = new int[3];
  - The following declarations are also equivalent:
  - char twod1[][] = new char[3][4];
  - char[][] twod2 = new char[3][4];
  - int[] nums, nums2, nums3; // create three arrays.
  - int nums[], nums2[], nums3[]; // create three arrays.