JAVA PROGRAMMING

UNIT - I

Dr Yogish H K, Professor, Dept. of ISE

Syllabus

- An Overview of Java: Introduction to Object-Oriented Programming, Simple Java Programs, identifiers, literals, Data Types, Variables, and Arrays: Java Is a Strongly Typed Language, The Primitive Types, Integers, Floating-Point Types, Characters, Booleans, Variables, Arrays, and Strings.

Introduction to Java

- Java is an object-oriented programming language developed by Sun Microsystems, initiated by **James Gosling**
- This language was initially called "Oak," but was renamed "Java" in 1995

Object Oriented Programming

- **Object** means a real world entity

  such as pen, chair, table etc.

- **Object-Oriented Programming**

  is a methodology or paradigm to design a program using classes and objects.

OOP Concepts

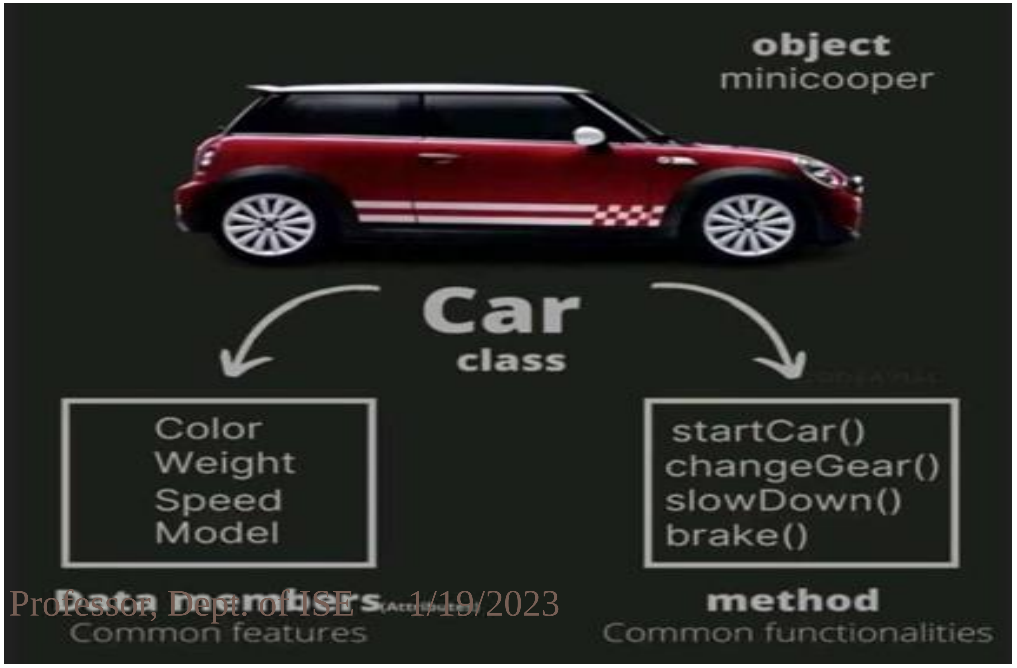
- It simplifies the software development and maintenance by providing some concepts:
  1. Object
  2. Class
  3. Inheritance
  4. Polymorphism
  5. Abstraction
  6. Encapsulation

Object..

- In simple words, object is a real word entity.
- For example, dog, car, pen, desk, etc. Each object shares the following two characteristics:
- **State:** The state of an object stored in **variables** (fields).
- **Behavior:** The **methods** shows the behavior of an object.
- Example of an Object – Dog
  - The **state** of dog includes, color, hungry, gender, breed, and age.
  - The dog **behaviour** includes running, barking, eating, sleeping, wagging tail, etc.

Class

- **Collection of objects** is called class.
- Class is not a real world entity. It is just a template or blueprint or prototype from which objects are created.
- Class does not occupy memory.
- Class is a group of variables of different data types and group of methods.
- A class in java can contain:
  - data /Variables
  - methods
  - constructors
  - nested class and
  - interfaces



Inheritance

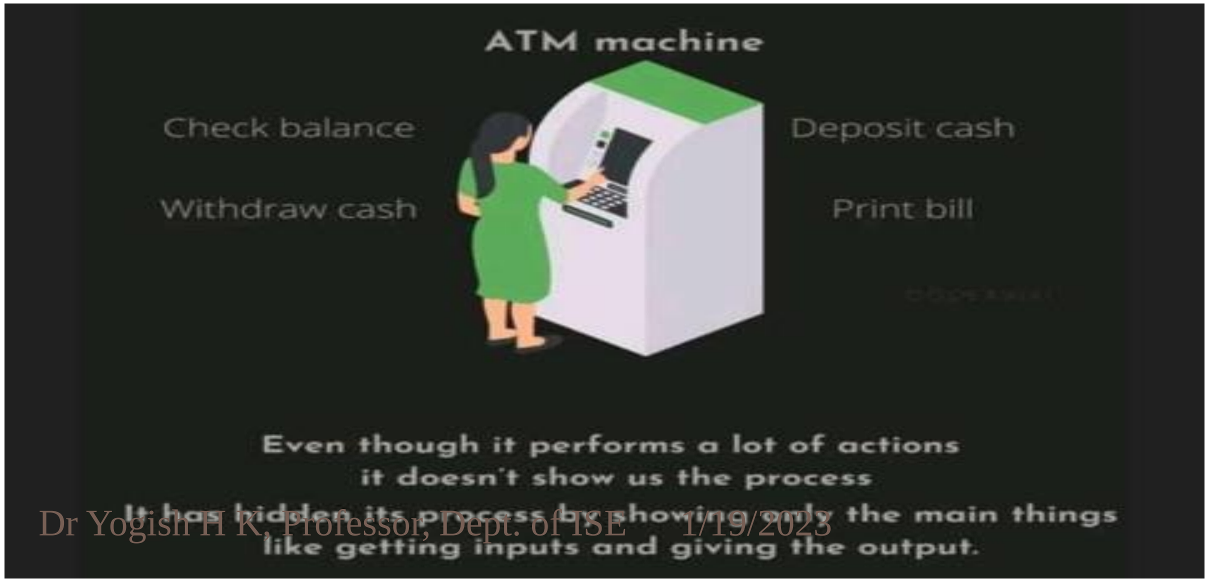
- Inheritance is the process by which one object acquires the properties of another object.(Variables and methods).
- In Java, inheritance means creating new classes based on existing ones.
- A class that inherits from another class can reuse the methods and variables of that class.
- In addition, you can add new variables and methods to your current class as well. It provides code reusability.

Polymorphism

- Polymorphism means "many forms“
- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
- Two types of polymorphism in Java: **compile-time polymorphism and runtime polymorphism.**
- Example : method overloading and method overriding.

Abstraction

- Hiding internal details and showing functionality is known as abstraction.
- For example: ATM machine, we don't know the internal processing.
- In java, we use abstract class and interface to achieve abstraction.

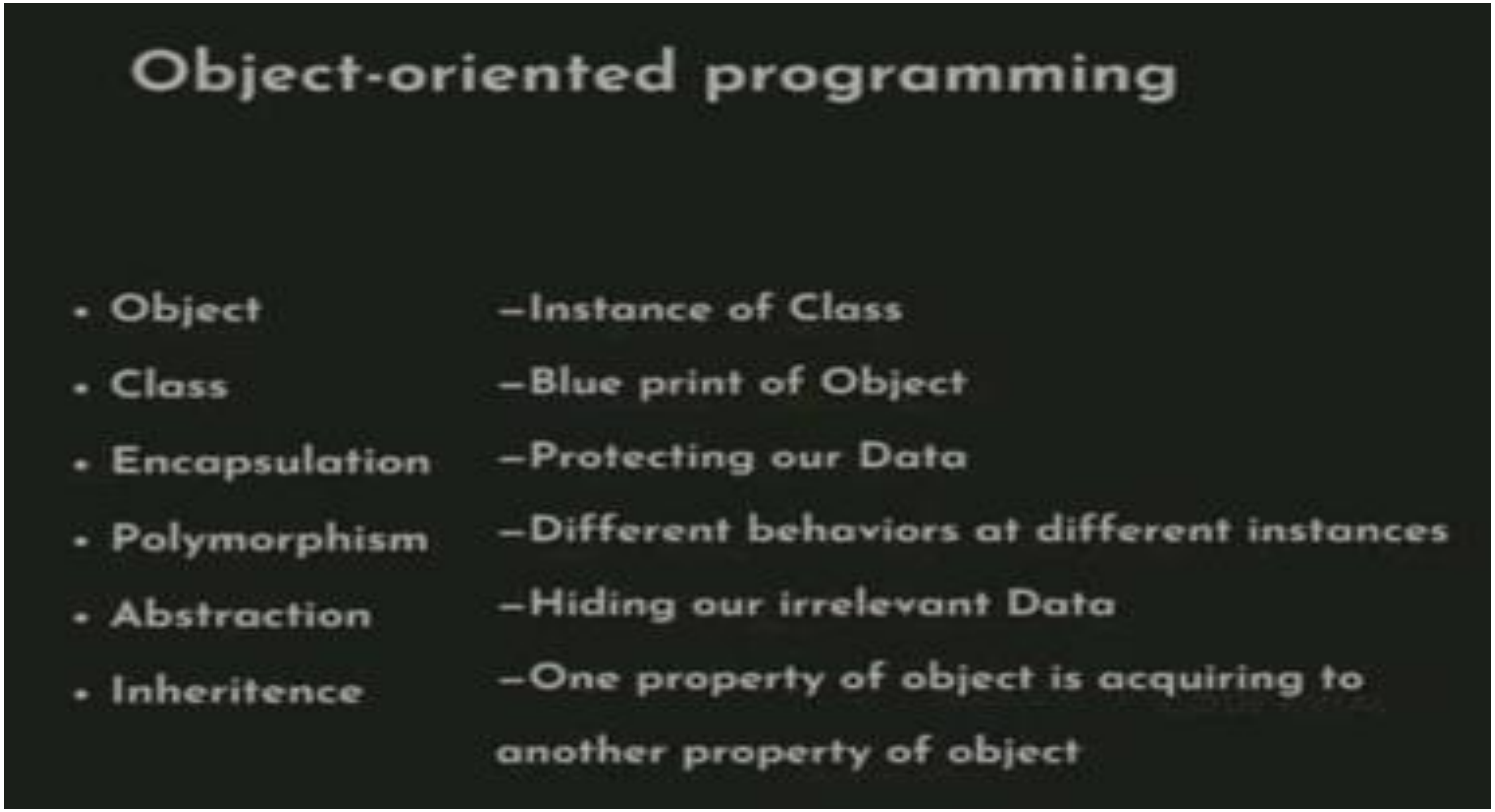

ATM machine
Check balance
Withdraw cash
Deposit cash
Print bill
Even though it performs a lot of actions
it doesn't show us the process
It has hidden its process by showing only the main things
like getting inputs and giving the output.

Encapsulation

- Binding (or wrapping) **code** and **data** together into a single unit is known as encapsulation.
  - For example: capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation.

School bag can keep our book,
pen, erasers, sharpner so on..

OOP Concepts …Finally



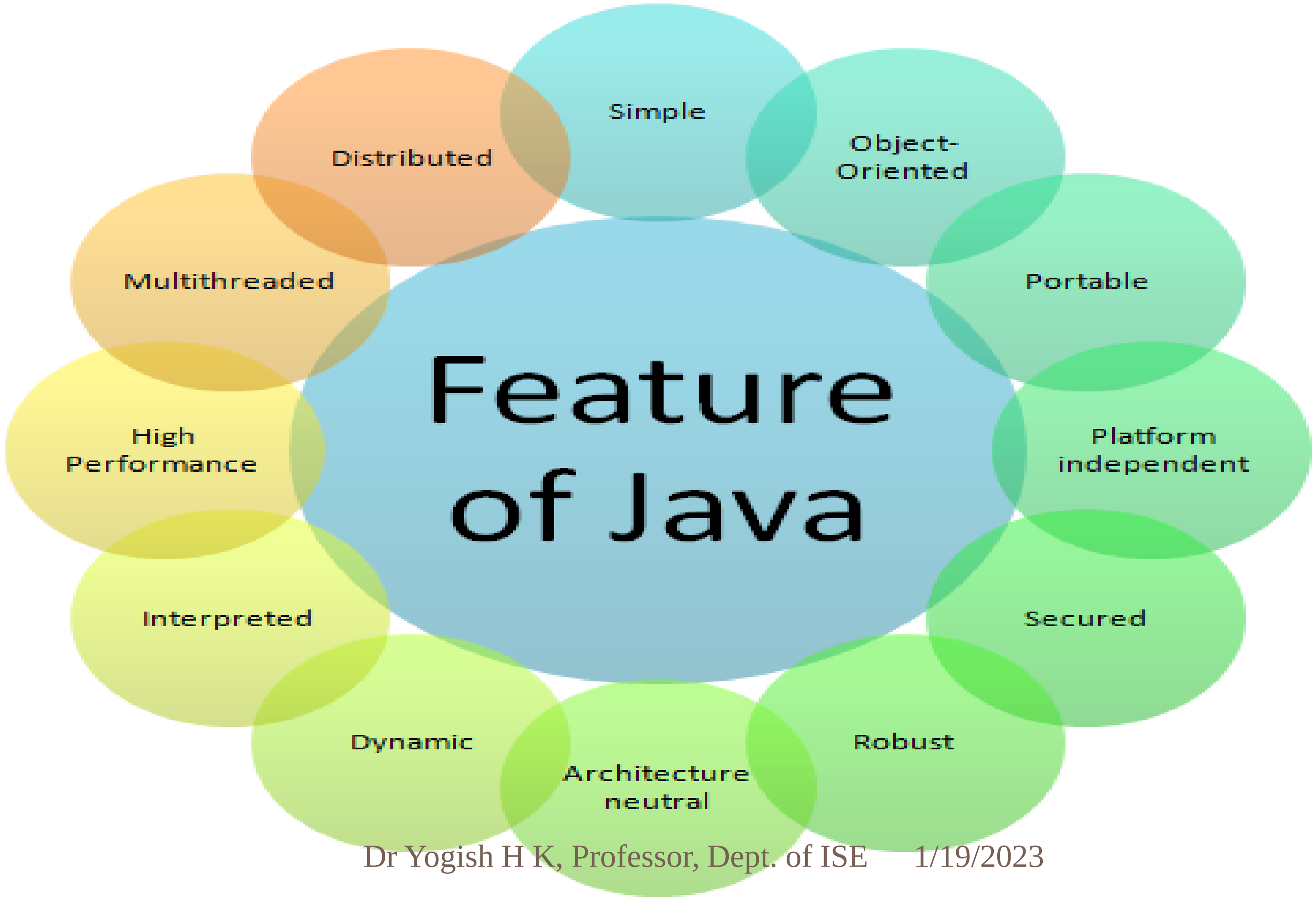
Object-oriented programming

• Object	–Instance of Class
• Class	–Blue print of Object
• Encapsulation	–Protecting our Data
• Polymorphism	–Different behaviors at different instances
• Abstraction	–Hiding our irrelevant Data
• Inheritence	–One property of object is acquiring to another property of object

Java better than C++ ?

- No Typedefs, Defines, or Preprocessor
- No Global Variables
- No Goto statements
- No Pointers
- No Unsafe Structures
- No Multiple Inheritance
- No Operator Overloading
- No Virtual Keyword

Java Buzzwords



Java Buzzwords

- **Simple**
  - Java is a simple Language because it contains many features of other Languages like C and C++ and removes complexity because it doesn’t support pointers, storage classes, goto statements and multiple inheritances.
- **Object Oriented**
  - Java is purely an Object Oriented Programming language i.e., all the code of the Java language is written into the classes and objects.

Java Buzzwords

- **Distributed**
  - Java is a distributed language because, because of its ability to share the data and programs over the LAN.
  - Access remote objects.

- **Multithreaded**
  - A Java program can be divided into multiple threads assigning different tasks for different threads and have all the threads executing in parallel.

Java Buzzwords

- **Dynamic**
  - The JVM maintains a lot of runtime information about the program and the objects in the program.
  - Libraries are dynamically linked during runtime.
- **Architecture Neutral**
  - Java follows “Write-once-run-anywhere” approach.
  - Java programs are compiled into byte code format which does not depend on any machine architecture but can be easily translated into a specific machine by a JVM for that machine.

Java Buzzwords

- **Portable**
  - In Java the size of the primitive data types are machine independent.
  - Int in Java is always a 32-bit integer, and float is always a 32-bit IEEE 754 floating point number.
- **Interpreted & High Performance**
  - Java programs are compiled to portable intermediate form known as byte codes, rather than to native machine level instructions and JVM executes the byte codes on any machine on which it is installed.
  - Just-in-time compiler.

Java Buzzwords

- **Robust**
  - A Program or an application is said to be robust (reliable) when it is able to give some response in any kind of context.
  - Java's features help to make the programs robust. Some of those features are: type checking, exception handling, etc.
- **Secured**
  - Java provides data security through encapsulation.

Java Buzzwords

- **Interpreted**
  - A Program or an application is said to be robust (reliable) when it is able to give some response in any kind of context.
  - Java’s features help to make the programs robust. Some of those features are: type checking, exception handling, etc.
- **Secured**
  - Java provides data security through encapsulation.

Types of Java Applications

- **Standalone Application**
  - It is also known as desktop application or window-based application.
  - An application that we need to install on every machine such as media player, antivirus etc.
  - AWT and Swing are used in java for creating standalone applications.

Types of Java Applications

- **Web Application**
  - An application that runs on the server side and creates dynamic page, is called web application.
  - Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

Types of Java Applications

- **Enterprise Application**
  - An application that is distributed in nature, such as banking applications etc.
  - It has the advantage of high level security, load balancing and clustering. I
  - n java, EJB is used for creating enterprise applications.
- **Mobile Application**
  - An application that is created for mobile devices.
  - Currently Android and Java ME are used for creating mobile applications.

First Java Program

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

First Java Program ….

- In Java, all code must reside inside a ***class* and the name of that class should match the name of the file that holds the program**.
- For this example, the name of the source file should be **HelloWorld.java**
- The Java compiler requires that a source file use the **.java** filename extension.

First Java Program …

- The keyword **'public'** says that, the **main( )** can be accessed even outside the defining class.
- The keyword **'static'** indicates that a function can be accessed without an object.
- The keyword **'void'** indicates that a function returns nothing.
- The Keyword **'main'** is the name of the function and entry point for a Java application.
- **String args [ ]** declares a parameter named args, which is an array of the class String. Objects of type String store character strings. **args** receives any command-line arguments present when the program is executed.

Variants of main() in Java

- Default:
  - **public static void main(String args[])**

- Variants:
  - **static** public void main(String args[])
  - public static void **main(String[] args**)
  - public **final** static void main(String[] args)
  - public **synchronized** static void main(String[] args)

Creating a Program

- Any text editor or Java IDE (Integrated Development Environment) can be used to develop Java programs
- Java source-code file names must end with the ***.java*** extension
- Some popular Java IDEs are
  - NetBeans
  - Eclipse
  - IntelliJ

Compiling / Running Java Program

- ***javac Welcome.java***
  - Searches the file in the current directory
  - Compiles the source file
  - Transforms the Java source code into bytecodes
  - Places the bytecodes in a file named **Welcome.class**
- **java Welcome**

Example2

```
/*    This is a simple Java program.
      Call this file "Example2.java".
*/
class Example2 {
// Your program begins with a call to main().
public static void main(String args[]){
System.out.println(“Good Morning");
  }
}
```

Print statements …

- You can concatenate arguments to println() with a (+) sign.

Example:

```
System.out.println("There are " + args.length + "
                                   command line arguments");
```

- Using print() instead of println() does not break the line.

For example,

```
System.out.print("There are ");
System.out.print(args.length);
System.out.print(" command line arguments");
```

Print Command line arguments

```
class PrintArgs {
            public static void main (String args[]) {
            for (int i = 0; i < args.length; i++) {
                System.out.println(args[i]);
            }
        }
    }
```

```
$ java PrintArgs Hello there!
Hello
there!
```

- System.out.println() prints its arguments followed by a linefeed (\n).

Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of Alphabets[a-z A-Z],Digits[0-9], the underscore(_) and dollar-sign characters.
- They must not begin with a Digits.
- Java is case-sensitive, so VALUE is a different identifier than Value.
- Some examples of valid identifiers are:
  - AvgTemp, count, A4, $count, Sum_of_digits
- Invalid Identifiers:
  - 4sum, sum-avg, yes/no

Literals

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

  100 , 120.5 , ‘x’ , “Good Morning”
- 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**.

Example3

```
/*          Here is another short example. Call this file "Example3.java".      */

class Example3 {
public static void main(String args[]) {
            int num;                        // this declares a variable called num
            num = 100;                      // this assigns num the value 100
            System.out.println("This is num: " + num);
            num = num * 2;
            System.out.print("The value of num * 2 is ");
            System.out.println(num);
     }
}
```

Output:

```
This is num: 100
The value of num * 2 is 200
```

Java Data Types - A data type, in programming, is a classification that specifies which type of value a variable has.

- Divided into two groups:

1. Primitive data types can also be called as simple types - byte, short, int, long, float, double, boolean and char.

- These can be put in four groups:
  - **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

2. Non-primitive data types – Arrays, Strings and Classes.

Java Data Types …

2. **Non-primitive data types** - String, Arrays and Classes.

Java Primitive Data Types

Data Type	Characteristics	Range
`byte`	8 bit signed integer	-128 to 127
`short`	16 bit signed integer	-32768 to 32767
`int`	32 bit signed integer	-2,147,483,648 to 2,147,483,647
`long`	64 bit signed integer	-9,223,372,036,854,775,808 to- 9,223,372,036,854,775,807
`float`	32 bit floating point number	$\pm$ 1.4E-45 to $\pm$ 3.4028235E+38
`double`	64 bit floating point number	$\pm$ 4.9E-324 to $\pm$ 1.7976931348623157E+308
`boolean`	`true` or `false`	NA, note Java booleans cannot be converted to or from other types
`char`	16 bit, Unicode	Unicode character, `\u0000` to `\uFFFF` Can mix with integer types

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.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.
- Its value is changing during program execution.
- **Declaring a Variable In Java**, all variables must be declared before they can be used.
  - The basic form of a variable declaration is shown here: **type identifier [= value][, identifier [= value] ...] ;**

Variables …

- Here are several examples of variable declarations of various types. Note that some include an initialization.

  - 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

- Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
- For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
class DynInit {
        public static void main(String args[]) {
            double a = 3.0, b = 4.0;
            double c = Math.sqrt(a * a + b * b);    //// c is dynamically initialized
            System.out.println("Hypotenuse is " + c);
    }
}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem).

Variables … Scope and Life time

- 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.
- Many other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java’s strict, object-oriented model.
- The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method’s scope.

Variables … Scope and Life time

- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation.
- Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

Variables … Scope and Lifetime

```
class Scope {                                   // Demonstrate block 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 … Scope and Lifetime

- Variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
- Therefore, variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.
- If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

Variables … Scope and Lifetime

```
// 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);
    }
  }
}
```

The output generated by this program is shown here:

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

Program to compute area of Circle

```
// Compute the area of a circle.
class Area
{
    public static void main(String args[])
    {
        double pi, r, area;
         r = 10.8; // radius of circle
         pi = 3.1416; // pi, approximately
         area = pi * r * r; // compute area
         System.out.println("Area of circle is " + a);
     }
}
```

Demonstrate char data type.

```
class CharDemo
{
        public static void main(String args[])
        {
                char ch1, ch2;
                ch1 = 88; // code for X
                ch2 = 'Y';
                System.out.print("ch1 and ch2: ");
                System.out.println(ch1 + "   " + ch2);
        }
}
```

- This program displays the following output:

```
ch1 and ch2: X   Y
```

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
```

Boolean

- Java has a primitive type, called boolean, for logical values.
- It can have only one of two possible values, true or false.
- This is the type returned by all relational operators.
- Variables are created using the keyword **boolean**.

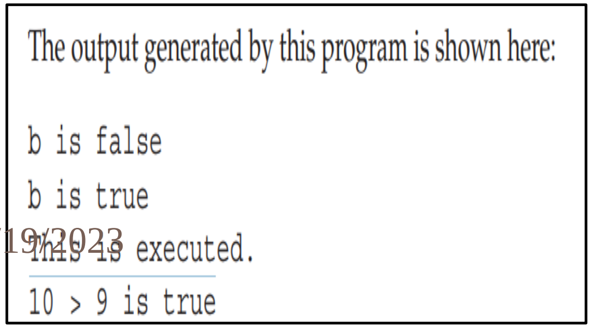
boolean b; // b stores either true or false value

we can assign the value for b;

b=false;

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);
        if(b)                 // a boolean value can control the if statement
                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
```

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.
- NOTE If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.

One-Dimensional Arrays

- A one-dimensional array is, essentially, a list of like-typed variables.
- To create an array, you first must create an array variable of the desired type.
- The general form of a one-dimensional array declaration is:

  type var-name[ ];
- Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold.
- For example, the following declares an array named month_days with the type “array of int”:
  - int month_days[];

Arrays …

- The value of month_days is set to null, which represents an array with no value.
- To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days.
- new is a special operator that allocates memory.
- The general form of new as it applies to one-dimensional arrays appears as follows:
  - array-var = new type[size];
- Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array.
- That is, 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. This example allocates a 12-element array of integers and links them to month_days.
- month_days = new int[12];
- After this statement executes, month_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Arrays …

- Let’s review: Obtaining an array is a two-step process
  - First, you must declare a variable of the desired array type.
  - Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.
- 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.
- For example, this statement assigns the value 28 to the second element of month_days. month_days[1] = 28;
- The next line displays the value stored at index 3. System.out.println(month_days[3]);

Demonstrate a one-dimensional array.

- Putting together all the pieces, here is a program that creates an array of the number of days in each month.

```
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 …

- When you run this program, it prints the number of days in April.
- As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.
- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:
  - **int month_days[] = new int[12];**

Array initialization

- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types.
- An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use new.
- For example, to store the number of days in each month, the following code creates an initialized array of integers: Animproved 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.");
      }
 }
```

Program to find average of set of numbers

- // Average an array of values.

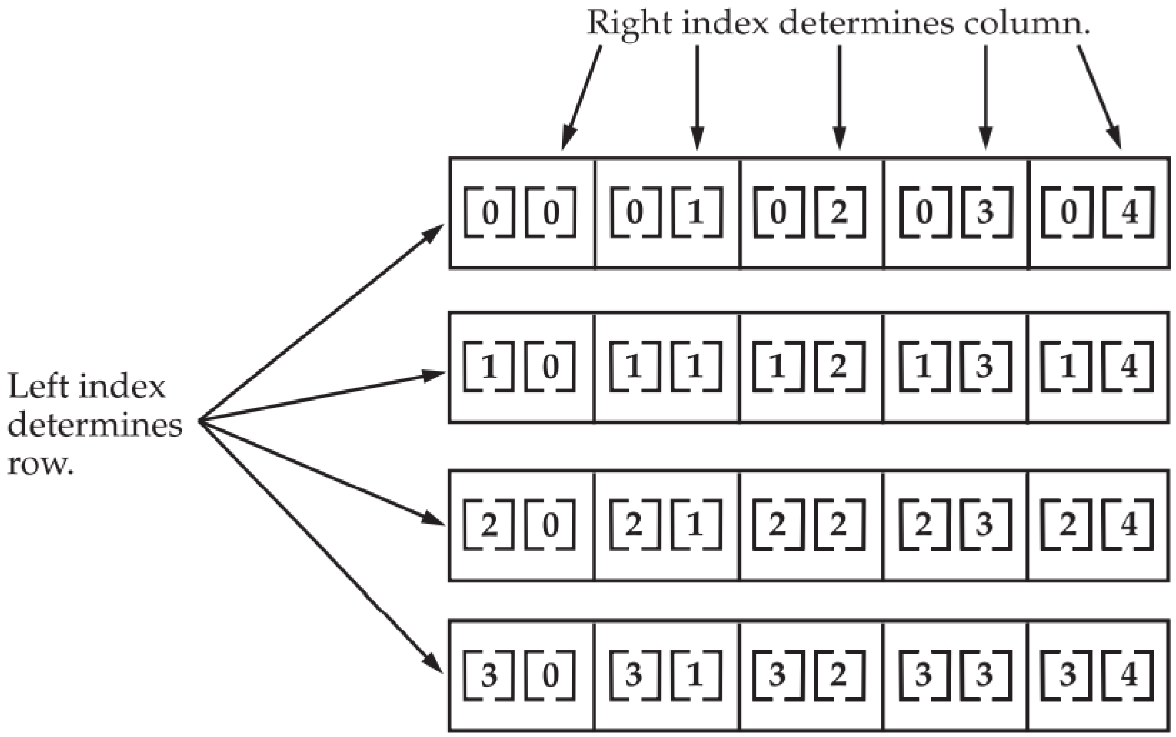
```
class Average {
        public static void main(String args[]) {
                double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
                double result = 0;
                int i;
                for(i=0; i<5; i++)
                        result = result + nums[i];
                System.out.println("Average is " + result / 5);
        }
}
```

Multidimensional Arrays

- In Java, multidimensional arrays are actually arrays of arrays. Ie. More than one dimension.
- For example, the following declares a two dimensional array variable called twoD.
- **int twoD[ ][ ] = new int[4][5];**
- This allocates a 4 by 5 array and assigns it to twoD.
- Internally this matrix is implemented as an array of arrays of int.

Multidimensional Arrays

- Conceptually, this array will look like the one shown in Figure.

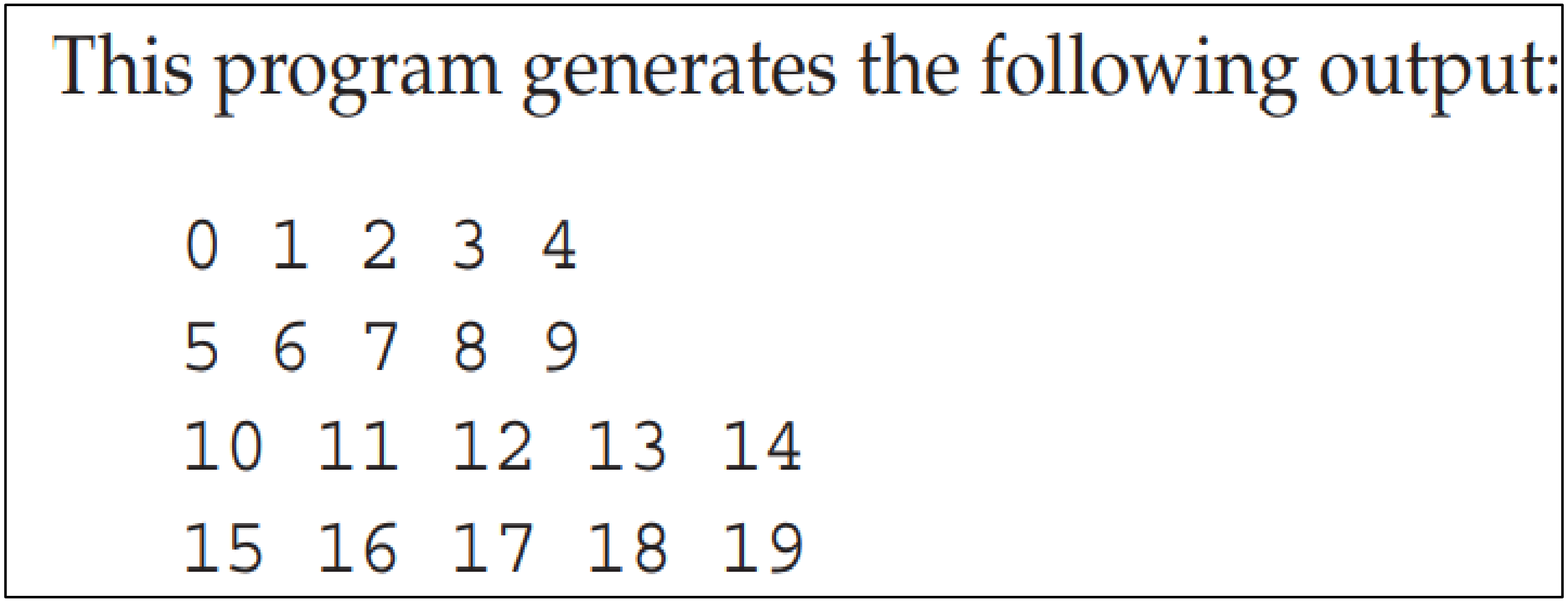


Given: int twoD [ ] [ ] = new int[4][5];

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();
                        }
      }
}
```

Output



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
```

Initialize multidimensional arrays

- Simply enclose each dimension’s initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.
- Also notice that you can use expressions as well as literal values inside of array initializers.

Initialize multidimensional arrays…

```
class Matrix {
        public static void main(String args[]) {
                int  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();
         }
    }
}
```

output

```
0     0      0     0
0     1     2     3
0     2     4     6
0     3     6     9
```

Alternative Array Declaration Syntax

- type[ ] var-name;
- The following two declarations are equivalent:
  - **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];**
- Declaring several arrays at the same time.
  - **int [] nums, nums2, nums3;** // create three arrays, creates //three array variables of type int. It is the same as writing
  - **int nums[], nums2[], nums3[];** // create three arrays

Strings

- String - array of characters, String defines an object.
- String type is used to declare string variables,
  - **String str;**
- A quoted string constant can be assigned to a String variable.
  - **str = “Dept of ISE";** or **String str = “Dept of ISE";**
- A variable of type String can be assigned to another variable of type String.
  - **String s;**
  - **s=str;**

Strings …

- Declares arrays of strings.

    String[] strar1;   //Declaration of String Array without size

    String[] strar2 = **new** String[2]; //Declaration with size

- String array Initialization:

1. String[] str1 = {"mon", "tue", " wed"};
2. String[] str2=**new** String[] {"mon", “tue", " wed"};
3. String[] str3= **new** String[3];

    str3[0]=“mon”;

    str3[1]=“tue”;

    str3[2]=“wed”;

Strings…

- You can use an object of type String as an argument to println( ).
- For example,
  - String str = "Dept of ISE";
  - System.out.println(str);
- Here, str is an object of type String. It is assigned the string “Dept of ISE”. This string is displayed by the println( ) statement.

Strings …

```
class mystr {
 public static void main(String[] args) {
  String []months= {"Jan", "Feb", "Mar", "April"};
  for (String i : months) {
          System.out.println(i);
      }
   }
}
```

THANK YOU