AKSHAYA INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi, Affiliated to VTU, Belagaavi, Recognised by GOK,
NBA Accredited (CSE)
Obalapura Post, Lingapura, Koratagere Road, Tumkur- 572 106, Karnataka

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Notes of Lessons

for

THIRD SEMESTER

on

"Object Oriented Programming with JAVA"

[BCS306A]

Prepared by:

DR. CHAMPAKAMALA S

Professor and Head

Department of Artificial Intelligence and Data Science

Akshaya Institute of Technology, Tumkur

Module 1

Session 1

Object-Oriented Programming:

Object-oriented programming (OOP) is at the core of Java. c. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

Two Paradigms

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed.

In procedural programming, the program is divided into small parts called ***functions***.	**In object-oriented programming**, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach***.	Object-oriented programming follows a ***bottom-up approach***.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.

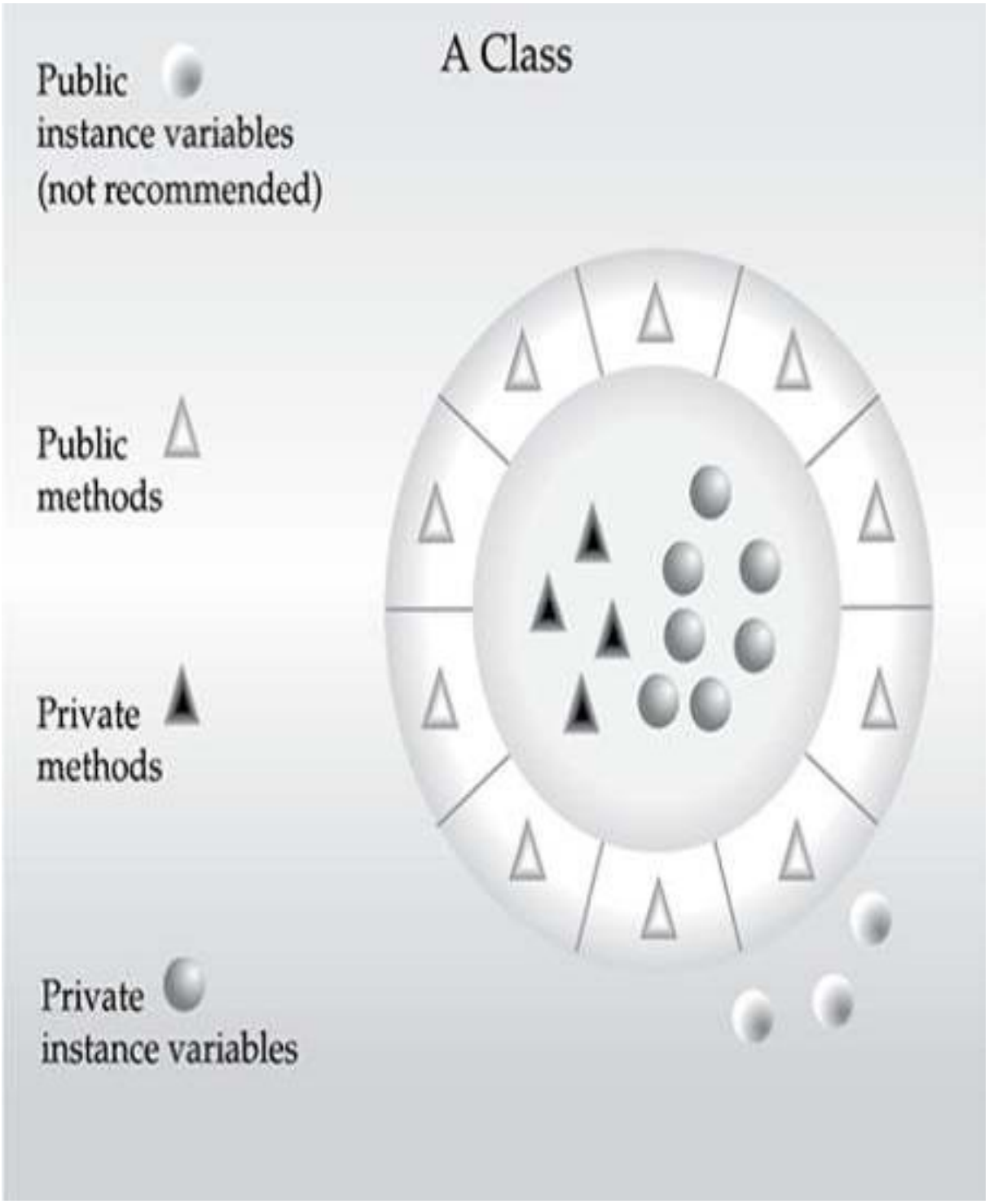
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.
Procedural programming is based on the ***unreal world***.	Object-oriented programming is based on the ***real world***.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

Abstraction:

Abstraction in Java is a process of hiding the implementation details from the user and showing only the functionality to the user. It can be achieved by using abstract classes, methods, and interfaces. An abstract class is a class that cannot be instantiated on its own and is meant to be inherited by concrete classes.

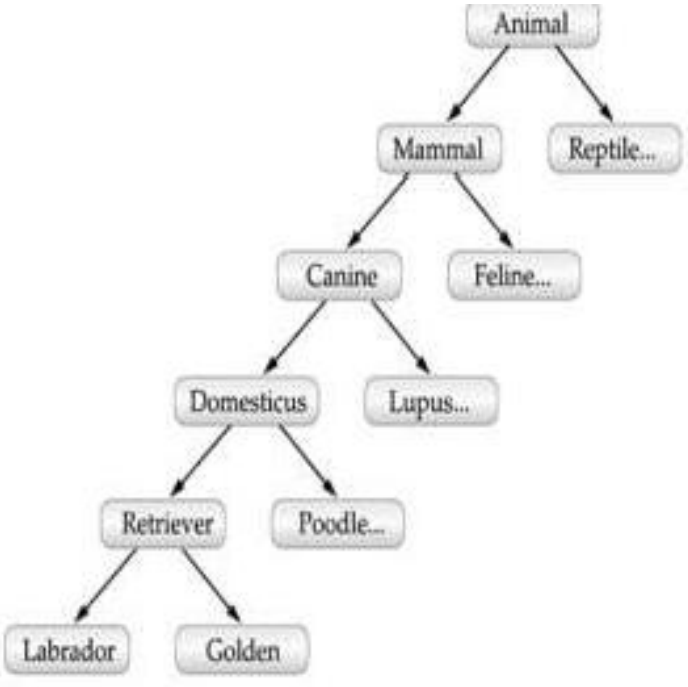
The Three OOP Principles:

1. **Encapsulation:** *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.



2. Inheritance

Inheritance is the process by which one object acquires the properties of another object.



3.Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions.

Lexical Issues

Whitespace

- Java is a free-form language. This means that you do not need to follow any special indentation rules.
- For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator.
- In Java, whitespace includes a space, tab, newline, or form feed.

Identifiers

- Identifiers are used to name things, such as classes, variables, and methods.
- An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar sign character is not intended for general use.)
- They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case sensitive, so **value** is a different identifier than value **.**

AvgTemp	count	a4	$test	this_is_ok

Invalid identifier names include these:

2count	high-temp	Not/ok

Literals

A constant value in Java is created by using a *literal* representation of it.For example:

100	98.6	'X'	"This is a test"

Comments:

There are the comment line,single line and multi line documentation comments.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is often used to terminate statements. The separators are shown in the following table:

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.
@	At-sign	Begins an annotation.

The Java Keywords

There are 67 keywords currently defined in the Java language.These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language.

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	non-sealed	open	opens
package	permits	private	protected	provides	public
record	requires	return	sealed	short	static
strictfp	super	switch	synchronized	this	throw
throws	to	transient	transitive	try	uses
var	void	volatile	while	with	yield
_					

Review Questions:

1. What is an object?
2. What are two types of programming approaches?
3. Mention three principles of oops with java?
4. What is abstraction?
5. Mention the lexical issues?

Session 2

Data Types, Variables, and Arrays

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean.**

• **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.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.

The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. The width and ranges of these integer types vary widely, as shown in this table.

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

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.

Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**: 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. Declaration:

short s;

short t;

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. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Although you might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case. The reason is that when **byte** and **short** values are used in an expression, they are *promoted* to **int** when the expression is evaluated.

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. For example, here is a program that computes the number of miles that light will travel in a specified number of days:

```
// Compute distance light travels using long variables.
class Light {
  public static void main(String[] args) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // approximate speed of light in miles per second
    lightspeed = 186000;

    days = 1000; // specify number of days here

    seconds = days * 24 * 60 * 60; // convert to seconds

    distance = lightspeed * seconds; // compute distance

    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
  }
}
```

This program generates the following output:

```
In 1000 days light will travel about 16070400000000 miles.
```

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

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

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. For example, **float** can be useful when representing dollars and cents.
Here are some example **float** variable declarations:
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 on 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. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice.
Here is a short program that uses **double** variables to compute the area of a circle:

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

   System.out.println("Area of circle is " + a);
  }
}
```

Characters

In Java, the data type used to store characters is **char**. A key point to understand is that Java uses *Unicode* to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.. At the time of Java's creation, Unicode required 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535. There are no negative **char**s.

Here is a program that demonstrates **char** variables:

```
// 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
```

Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter *X*. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set.

Although **char** is designed to hold Unicode characters, it can also be used as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable. Consider the following program:

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

Booleans

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, 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**.
Here is a program that demonstrates the **boolean** type

```
// 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 valu
    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. In addition, all variables have a scope, which defines their visibility, and a lifetime.

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*] …];

Here, *type* is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list.
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'.
```

Type Conversion and Casting

Java's Automatic Conversions

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.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

Casting Incompatible Types

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:
(*target-type*) *value*
Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**.

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.
int a; byte b; // … b = (byte) a;
A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional

components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range. The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
  public static void main(String[] args) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
```

Conversion of double to int.

d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

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. To handle this kind of problem, 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**.

As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type. In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

Review Questions:

1. What are data types?
2. What are variables?
3. What is type casting and type conversion?
4. Mention two types of conversions?
5. Give the syntax for declaring a variable?
6. Define scope of variables?

Session 3

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.

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 element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element 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;
```

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 (for numeric types), **false** (for **boolean**), or **null**.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.

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. If the concept of dynamic allocation is unfamiliar to you, don't worry. It will be described at length later in this book.

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]);

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

```
// 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.");
  }
}
```

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];
```

Multidimensional Arrays

In Java, *multidimensional arrays* are implemented as arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. 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**. Conceptually, this array will look like the one shown in figure:

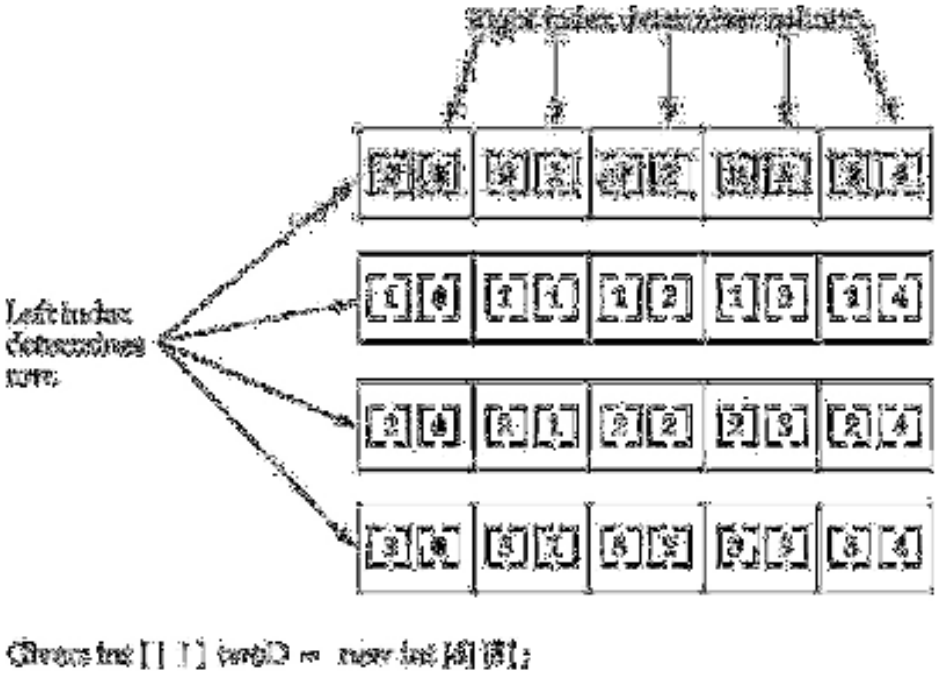


FIGURE: A conceptual view of a 4 by 5, two-dimensional array

Introduction Type Inference with Local Variables

A new feature called *local variable type inference* was added to the Java language. To begin, let's review two important aspects of variables. First, all variables in Java must be declared prior to their use. Second, a variable can be initialized with a value when it is declared. Furthermore, when a variable is initialized, the type of the initializer must be the same as (or convertible to) the declared type of the variable. Thus, in principle, it would not be necessary to specify an explicit type for an initialized variable because it could be inferred by the type of its initializer. Of course, in the past, such inference was not supported, and all variables required an explicitly declared type, whether they were initialized or not.

To use local variable type inference, the variable must be declared with **var** as the type name and it must include an initializer. For example, in the past you would declare a local **double** variable called **avg** that is initialized with the value 10.0, as shown here:

```
double avg = 10.0;
```

Using type inference, this declaration can now also be written like this:

```
var avg = 10.0;
```

In both cases, **avg** will be of type **double**. In the first case, its type is explicitly specified. In the second, its type is inferred as **double** because the initializer 10.0 is of type **double**.

As mentioned, **var** is context-sensitive. When it is used as the type name in the context of a local variable declaration, it tells the compiler to use type inference to determine the type of the variable being declared based on the type of the initializer. Thus, in a local variable declaration, **var** is a placeholder for the actual, inferred type. However, when used in most other places, **var** is simply a user-defined identifier with no special meaning. For example, the following declaration is still valid:

```
int var = 1; // In this case, var is simply a user-defined
identifier.
```

In this case, the type is explicitly specified as **int** and **var** is the name of the variable being declared. Even though it is context-sensitive, there are a few places in which the use of **var** is illegal. It cannot be used as the name of a class, for example.

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String[] args) {

    // Use type inference to determine the type of the
    // variable named avg. In this case, double is inferred.
    var avg = 10.0;
    System.out.println("Value of avg: " + avg);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);

    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

Here is the output:

```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

The preceding example uses **var** to declare only simple variables, but you can also use **var** to declare an array. For example:

```
var myArray = new int[10]; // This is valid.
```

Notice that neither **var** nor **myArray** has brackets. Instead, the type of **myArray** is inferred to be **int[ ]**. Furthermore, you *cannot* use brackets on the left side of a **var** declaration. Thus, both of these declarations are invalid:

Operators

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators.

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

Program to demonstrate different type of operators

```
//This program explains about operator
public class Operator {
        public static void main(String[] args) {
                int a=10,b=20;
                System.out.println(a==b);
                System.out.println(a!=b);
                System.out.println(a>b);
                System.out.println(a<b);
```

```
                System.out.println(a<=b);
                System.out.println(a>=b);
                System.out.println(a==b && a!=b);
                System.out.println(a==b || a!=b);
                System.out.println(!(a>b));
                System.out.println(a--);
                System.out.println(++a);
                System.out.println(a++);
                System.out.println(++a);
        }
}
```

Arithmetic Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment

a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

This version uses the += compound assignment operator. Both statements perform the same action. There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form

var = var op expression; can be rewritten as var op= expression;

The compound assignment operators provide two benefits. First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, in some cases they are more efficient than are their equivalent long forms.

Here is a sample program that shows several op = assignments in action:

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Review questions:

1. Define Arrays?
2. Mention two types of arrays?
3. What is variable type inference?
4. Mention different arithmetic operators?
5. What is assignment operator?

Session 4

Relational Operators

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here

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

The outcome of these operations is a Boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements

Boolean Logical Operators

The Boolean logical operators shown here operate only on Boolean operands. All of the binary logical operators combine two Boolean values to form a resultant Boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment

Operator	Result
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String[] args) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("         a = " + a);
    System.out.println("         b = " + b);
    System.out.println("       a|b = " + c);
    System.out.println("       a&b = " + d);
    System.out.println("       a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("        !a = " + g);
  }
}
```

The output of the above program is:

```
          a = true
          b = false
        a|b = true
        a&b = false
        a^b = true
!a&b|a&!b = true
         !a = false
```

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. These are secondary versions of the Boolean AND and OR operators, and are commonly known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it: if (denom != 0 && num / denom > 10) Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would be evaluated, causing a run-time exception when demon is zero

The? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?.

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.

```
// Demonstrate ?.
class Ternary {
  public static void main(String[] args) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

Table shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left).

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	*(type-cast)*
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	op=					
Lowest						

Review Questions:

1. Give the different relational operators?
2. What are Boolean logical operators?
3. What is operator precedence?
4. Define ternary operator?
5. Define operator parentheses?

Session 5

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories:

1. Selection
2. iteration
3. jump.

Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).

Jump statements allow your program to execute in a nonlinear fashion

Java's Selection Statements

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time

If

if (condition) statement1;

else statement2;

```
boolean dataAvailable;
//...
if (dataAvailable)
  ProcessData();
else
  waitForMoreData();
```

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

if(condition)

statement;

else if(condition)

statement;

else if(condition)

statement;

.

.

.

else

statement

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String[] args) {
  int month = 4; // April
  String season;

  if(month == 12 || month == 1 || month == 2)
    season = "Winter";
  else if(month == 3 || month == 4 || month == 5)
    season = "Spring";
  else if(month == 6 || month == 7 || month == 8)
    season = "Summer";
  else if(month == 9 || month == 10 || month == 11)
    season = "Autumn";
  else
    season = "Bogus Month";

  System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

The Traditional switch

The switch statement is Java's multi way branch statement. It provides an easyway to dispatch execution to different parts of your code based on the value of an expression.

```
switch (expression) {
  case value1:
     // statement sequence
     break;
  case value2:
     // statement sequence
     break;
.
.
.
  case valueN :
     // statement sequence
     break;
  default:
    // default statement sequence
}
```

The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String[] args) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

while

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
 // body of loop
}
```

Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstrate the while loop.
class While {
  public static void main(String[] args) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

Review questions:

1. Mention the different selection statement?
2. What is traditional switch?
3. What are iteration statements?
4. Give the general form of while statements?
5. Give the General form of traditional switch statement?

Session 6

do-while

As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
 // body of loop
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression

```
// Demonstrate the do-while loop.
class DoWhile {
  public static void main(String[] args) {
    int n = 10;

    do {
      System.out.println("tick " + n);
      n--;
    } while(n > 0);
  }
}
```

For

for(initialization; condition; iteration) {

// body

}

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String[] args) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

```
// Using the comma.
class Comma {
  public static void main(String[] args) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

The For-Each Version of the for Loop

A second form of for implements a "for-each" style loop. As you may know, contemporary language theory has embraced the for-each concept, and it has become a standard feature that programmers have come to expect. A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. In Java, the for-each style of for is also referred to as the enhanced for loop. The general form of the for-each version of the for is shown here for (type itr-var : collection) statement-block Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String[] args) {
    int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;

    // use for-each style for to display and sum the values
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }

    System.out.println("Summation: " + sum);
  }
}
```

The output from the program is shown here:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Iterating Over Multidimensional Arrays

```
// Use for-each style for on a two-dimensional array.
class ForEach3 {
  public static void main(String[] args) {
    int sum = 0;
    int[][] nums = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // use for-each for to display and sum the values
    for(int[] x : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String[] args) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action

```
// Demonstrate continue.
class Continue {
  public static void main(String[] args) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

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

Return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```
// Demonstrate return.
class Return {
  public static void main(String[] args) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Review questions:

1. Give the general form for do while?
2. Give the general form for traditional for statement?
3. Mention the different jump statement?

MODULE-1

1. Explain how Object Oriented Programming model is different from Process/procedure driven programming model?
2. Define the term ‘Abstraction` and how ‘abstraction’ applied to traditional computer programs?
3. Explain three OOP principles: Encapsulation, Inheritance and Polymorphism.
4. Explain the following with a suitable JAVA program segment.
    a. Code blocks
    b. Comments
    c. Identifiers
    d. Literals
5. Explain integer and Boolean data types in Java and their relevance in programming.
6. Explain floating point and character data types in Java and their usage in problem solving.
7. Define variable and dynamic initialization variables with syntax and examples in Java.
8. Define scope, scope types and lifetime of variables with respect to Java with suitable examples.
9. Explain automatic type conversion and explicit type conversion support in Java with suitable examples.
10. Explain automatic type promotion scenarios in Java with examples.
11. Define array. Explain one dimensional array declaration and element access in Java with examples.
12. Explain one dimensional array initialization with examples.
13. Explain declaration and initialization of multi-dimensional arrays in Java with code segments.
14. Explain the term ‘type inference’. Explain Java support and rules for type inference with suitable examples.
15. Develop a Java program to compute the sum of three digit number.
16. Develop a program to create/initialize a Jagged array containing Floyd’s triangle.
17. Explain modulus and compound assignment operators with suitable Java code segments.
18. Explain the use of increment and decrement operators with suitable Java code segments.
19. List and explain, the role of relational operators in programming.
20. Explain the role of short-circuit logical operators in programming with suitable Java code segments.
21. With syntax and example, explain the ternary (?) operator.
22. Develop a program to test the given number is odd or even.
23. Develop a program to the given number is positive and divisible by 5.
24. Explain with syntax and example, the structure if, nested if and if-else-if ladder.
25. Develop a program to find maximum among three numbers.

26. Develop a program to input electricity unit charges and calculate total electricity bill according to the given condition:

    For first 50 units Rs. 0.50/unit

    For next 100 units Rs. 0.75/unit

    For next 100 units Rs. 1.20/unit

    For unit above 250 Rs. 1.50/unit

    An additional surcharge of 20% is added to the bill

27. Develop a program to find all roots of a quadratic equation.
28. Explain traditional switch statement with an example. Highlight the role of break and default in switch statement.
29. Develop a program to create Simple Calculator (+, -, x, /) using switch-case.
30. Develop a program print total number of days in a month using switch case.
31. Explain while and do-while statement with syntax and example. Highlight the similarity and difference between while and do-while statement.
32. Develop a program to print Fibonacci series up to n terms.
33. Develop a program to check a given number is Armstrong or not.
34. Explain traditional for statement with syntax and example.
35. Develop a program to print all Perfect numbers between 1 to n.
36. Develop a program to print Pascal triangle upto n rows.
37. Explain for-each (enhance for) statement with syntax and example.
38. Develop a program to find sum and average of N items present in a array (use for-each).
39. Develop a program to create a matrix, print the matrix and find the row-sum (use for-each).
40. Explain the role of continue statement with an example.
41. Develop a program to find the sum of even numbers present in an array (use continue statement).
42. Explain the use of break statement, to exit a loop and as civilized goto statement with suitable code segments/examples.
43. Develop a program to check a given number is prime or not (use break statement).
44. Explain the role of 'return' in Java with an example.
45. Develop a program to find the value of a/b. terminate the program on the value of b=0;
46. Develop a program to find a/b. Use guard expression (short-circuit operator) for b!=0.