Dayananda Sagar College of Engineering

Department of Information Science and Engineering

Department of Information Science and Engineering

DSCE

Dept. of Information Science & Engg

Course Name: INTRODUCTION TO JAVA

Course Code:18IS6IEJVA

Semester:6

Faculty : Mrs.Radhika T V
Assistant Professor , Dept of ISE
DSCE

Chapter 1:
INTRODUCTION

Creation of Java

- Java was invented by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan (Green Team) at Sun Microsystems, Inc. in 1991.
- Initially designed for small, embedded systems in electronic appliances like set-top boxes.
- The language was initially called “Oak,” but was renamed “Java” in 1995.
- The primary motivation was the need for a platform-independent (that is, architecture- neutral)language.
- The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target.

Creation of Java(Cntd.)

- With the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.
  - Internet ultimately led to Java's large-scale success.

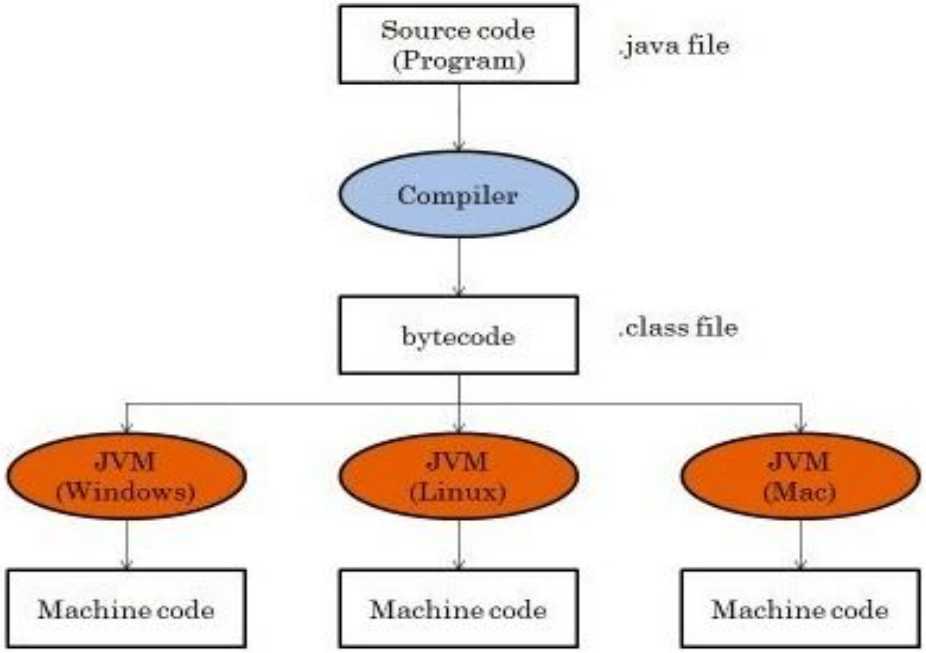
- Java shares some of the other attributes that helped make C and C++ successful.
  - Java is a programmer's language.
  - Java is cohesive and logically consistent.
  - Java gives the programmer full control

- While it is true that Java was influenced by C++, it is not an enhanced version of C++.

BYTECODE

- Bytecode (.class file) 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).

- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments
  - The fact that a Java program is executed by the JVM also helps to make it secure.



Advantage of Java Bytecode

- Platform independence

- Add portability to Java which is lacking in languages like C or C++.

- Sun Microsystems captioned JAVA as "write once, read anywhere" or "WORA" in resonance to the bytecode interpretation.

The Java Buzzwords

- **Simple**
  - Java inherits the C/C++ syntax and many of the object-oriented features of C++.
- **Secure**
  - Allowing Java programs to be executed by the JVM makes Java program fully secured under the control of the JVM.
- **Portable**
  - Ability to run the program on any platform and no dependency on the underlying hardware / operating system.
- **Object-oriented**
  - The object oriented model in Java is simple and easy to extend

The Java Buzzwords(cntd..)

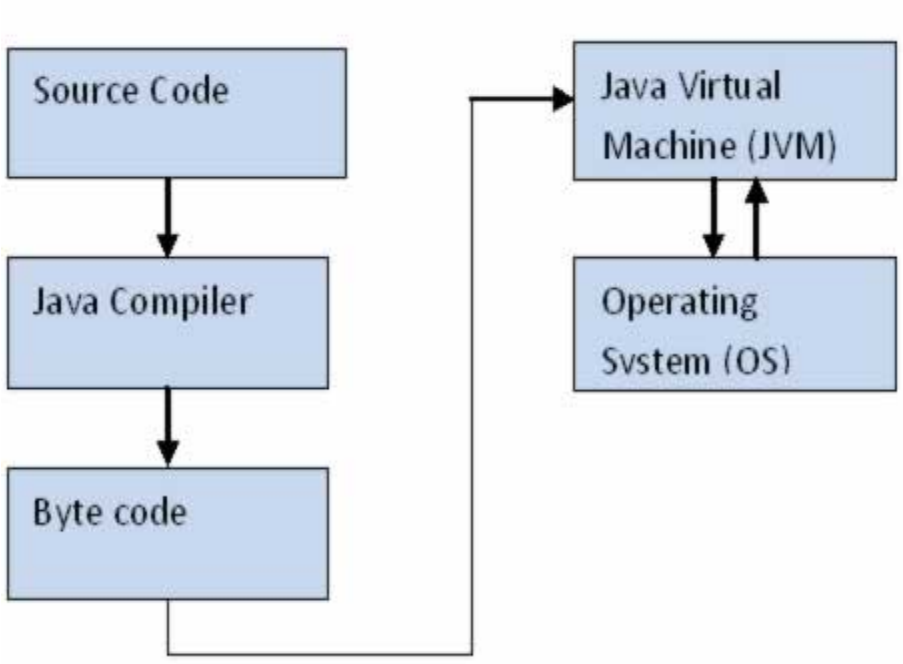
- **Robust**
  - Platform Independent:
  - Object Oriented Programming Language:
  - Memory Management:
  - Exception Handling:
- **Multithreaded**
  - Allows to develop program that can do multiple task simultaneously.
- **Architecture-neutral**
  - Java code does not depend on the underlying architecture and only depends on it JVM

The Java Buzzwords(cntd..)

- **Interpreted**
  - JVM interprets the ByteCode into Machine instructions during runtime.
- **High performance**
  - JVM was intelligently developed to interpret only the piece of the code that is required to execute and untouch the rest of the code
- **Distributed**
  - Java has a feature called Remote Method Invocation (RMI)
- **Dynamic**
  - Java programs access various runtime libraries and information

Java Architecture

Compilation and interpretation in Java



Object-Oriented Programming

- Everything in java is an object.
- **Two Paradigms**
  - ***process-oriented model.***
  - ***object-oriented programming,***

Abstraction

- Hiding implementation details from user and providing only the functionality.
- An essential element of object-oriented programming is *abstraction.*
- Humans manage complexity through abstraction.
- For example, people do not think of a car as a set of tens of thousands of individual parts- ***car is a single object***
- Abstraction lets you focus on what the object does instead of how it does it.

Abstraction (cntd..)

- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.

- There are two ways to achieve abstraction in java
  - Abstract class
  - Interface

The Three OOP Principles

1. **Encapsulation**
2. **Inheritance**
3. **Polymorphism**

Encapsulation

- The process of binding data and corresponding methods (behavior) together into a single unit is called **encapsulation in Java**.

- Encapsulation is a programming technique that binds the class members together and prevents them from being accessed by other classes.

- Every Java class is an example of encapsulation

- Another example of encapsulation is a capsule.
- Other examples are school bag, login to gmail account etc.

Encapsulation(cntd..)

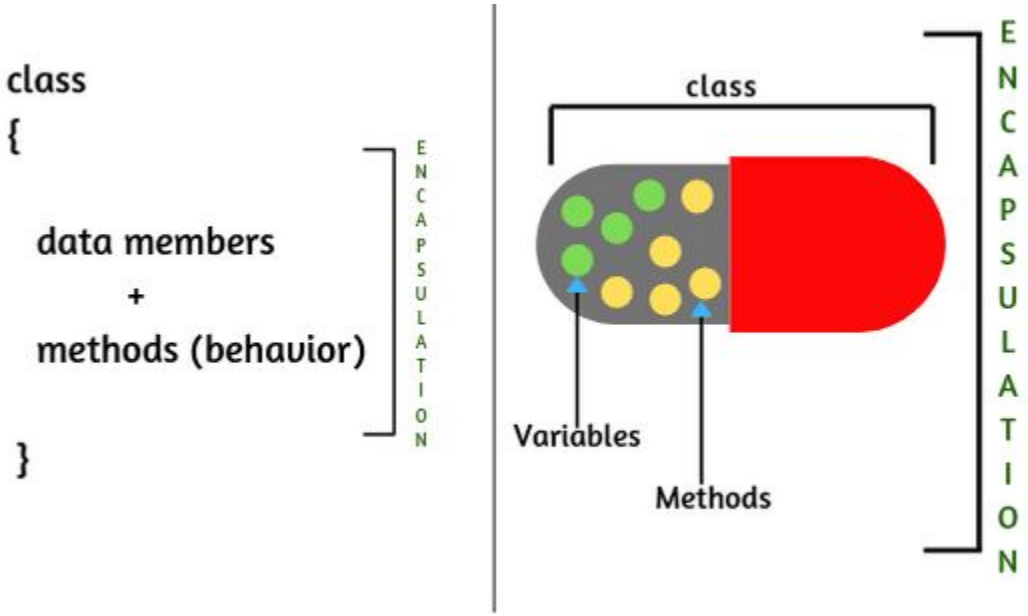


Fig: Encapsulation

Inheritance in Java

- The process of obtaining the data members and methods from one class to another class is known as **inheritance**.
- It is one of the fundamental features of object-oriented programming.
- **Important points**
  - Class which is give data members is known as **base or super or parent class.**
  - Class which is taking the data members and methods is known as **sub or derived or child class.**

Inheritance in Java

- The concept of inheritance is also known as re-usability .

Use of Inheritance

- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism
- For code Re-usability

Syntax of Inheritance

class Subclass-Name **extends** Superclass-Name

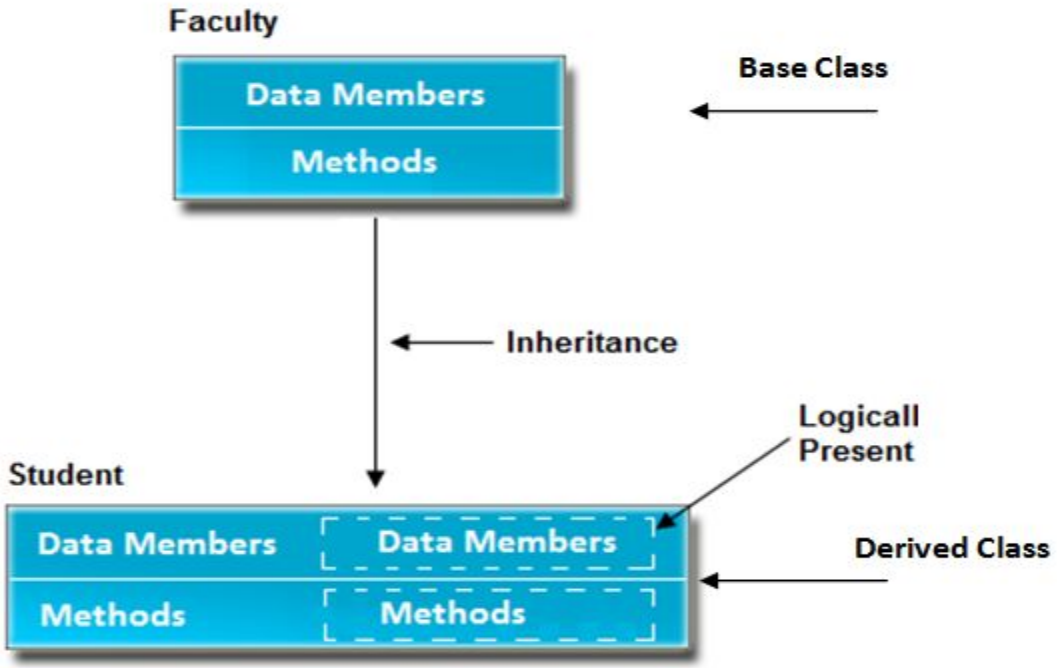
{

//methods and fields

}

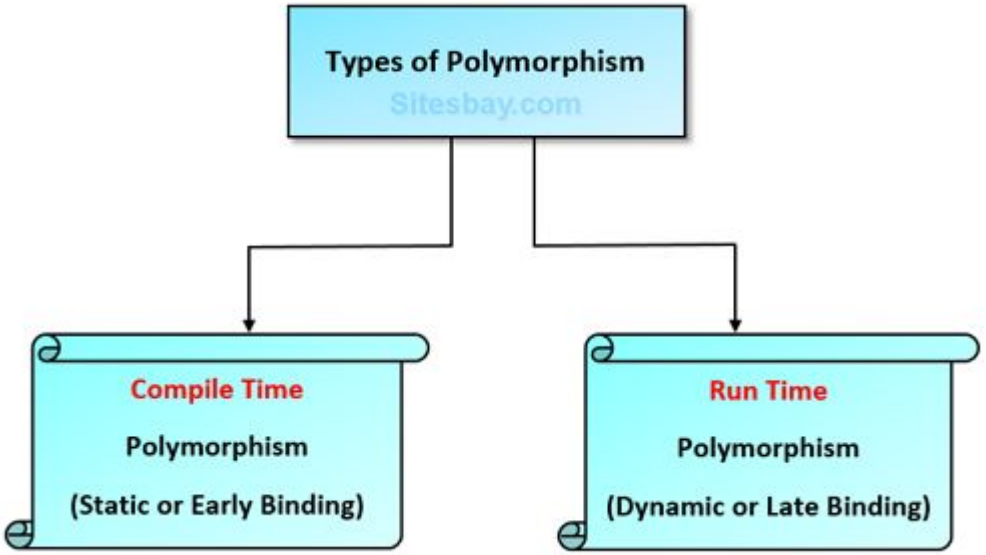
Inheritance in Java(cntd..)

- The real life example of inheritance is child and parents, all the properties of father are inherited by his son
- Another example is illustrated in diagram below



Polymorphism

- The process of representing one form in multiple forms is known as Polymorphism
- Polymorphism is derived from 2 greek words: poly and morphs.
- Polymorphism is not a programming concept but it is one of the principal of OOPs.



Real life example of polymorphism in Java

DSCE
Dept. of Information Science & Engg

In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

Real life example of polymorphism in Java (cntd..)

A First Simple Program


```
/*
This is a simple Java program. Call this file
    "Example.java".
*/

    class Example {
// Your program begins with a call to main(). public static
    void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Steps of Execution

- **Save the filename with classname. Here it is Example.java**
- **Compiling:**

  C:\>javac Example.java
- The **javac**compiler creates a file called **Example.class** that contains the bytecode version of the program.
- To actually run the program, you must use the Java application launcher, called **java**.

  C:\>java Example
- When the program is run, the following output is displayed:

  This is a simple Java program.

Example 2

DSCE
Dept. of Information Science & Engg

```
/*
Here is another short example. Call this file "Example2.java".
*/
class Example2 {
public static void main(String args[])
{ int num;
num = 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: When you run this program, you will see the following output:

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

Two Control Statements

DSCE
Dept. of Information Science & Engg

The if Statement

Syntax: **if(*condition*) *statement*;**

- Here, *condition* is a Boolean expression
- **Example:** if(num< 100)

```
System.out.println("num is less than 100");
```

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

Program that illustrates the if statement:


```
/* Demonstrate the if. Call this file "IfSample.java".*/
Class IfSample {
    public static void main(String args[])
{     int x, y;
      x = 10;
      y = 20;
if(x < y)
      System.out.println("x is less than y");
x = x * 2;
if(x == y)
      System.out.println("x now equal to y");
x = x * 2;
if(x > y) System.out.println("x now greater than y");
// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```

The output :

```
x is less than y
x now equal to y
x now greater than y
```

The for Loop

- **Syntax:**

for(initialization; condition; iteration) statement;

- The initialization portion of the loop sets a loop control variable to an initial value.
- The condition is a Boolean expression that tests the loop control variable

Program that illustrates the for statement:


```
/*
Demonstrate the for loop. Call this file "ForTest.java".
*/
Class ForTest {
public static void main(String args[])
{
 int x;
for(x = 0; x<10; x = x+1)
 System.out.println("This is x: " + x);
}
}
```

The output :

```
This is x: 0
 This is x: 1
This is x: 2
This is x: 3
 This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
y
```

Lexical Issues

1.Whitespace

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- In Java, whitespace is a space, tab, or newline

2. Identifiers

- Identifiers are used for class names, method names, and variable names.
- It is descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.
- They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Example: *Valid identifiers are*

AvgTemp count, a4,$test,this_is_ok

Invalid identifiers are

2Count, high-temp, Not/ok

Lexical Issues

DSCE
Dept. of Information Science & Engg

3. 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“ int a=100;

  char d=“x”

4.Comments

There are three types of comments defined by Java

- Single line
- Multiline
- Documentation: This type of comment is used to produce an HTMLfile that documents your program. The documentation comment begins with a /** and ends with a */.

Lexical Issues

5. Separators

In Java, there are a few characters that are used as separators.

Symbol Name	Purpose
( )	Used to contain lists of parameters in method definition and invocation.
{ }	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Used to declare array types. Also used when dereferencing array values.
;	Terminates statements.
, int a=0, b=8,c;	Separates consecutive identifiers in a variable declaration.
. Import java.util.*;	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Primitive Types

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

DSCE
Dept. of Information Science & Engg

- Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values.

Name	Width	Range	Example
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long days ; long seconds;
int	32	–2,147,483,648 to 2,147,483,647	int a
short	16	–32,768 to 32,767	short s; short t;
byte	8	–128 to 127	byte b, c;

Floating-Point Types

- Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision.

- Example: calculations such as square root. O.4567 X 10_2
- Java implements the standard (IEEE–754) set of floating-point types and operators.
- There are two kinds of floating-point types, **float** and **double**,

Name	Width in Bits	Approximate Range	Example
Double	64	4.9e–324 to 1.8e+308	double a;
float	32	1.4e–045 to 3.4e+038	Float hightemp, lowtemp;

Characters

- In Java, the data type used to store characters is **char**.
- **char** in Java is not the same as **char** in C or C++.
- In C/C++, **char** is 8 bits wide.
- 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. The range of a **char** is 0 to 65,536.

Characters (cntd..)

- **Program that demonstrates char variables**:

```
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);
}
}
```

Output:

ch1 and ch2: X Y

Boolean

- Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**.

Program:

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

Output:

```
b is false
 b is true
This is executed.
```

Variables

- A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

- All variables must be declared before they can be used.
- ***Syntax*** *: type identifier* [= *value*];
- Example: int a, b, c; // declares three ints, a, b, and c.
- int d = 3, e, f = 5; // declares three more ints, initializing e and f

Dynamic Initialization

- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

Variables(cntd..)


```
// Program to 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);
}
}
```

Type Conversion and Casting

- Assigning a value of one type to a variable of another type is possible .
- If the two types are compatible, then Java will perform the conversion automatically.
- **For example,** it is always possible to assign an **int** value to a **long** variable.
- But not all types are compatible, and thus, not all type conversions are **implicitly** allowed. For instance, there is no automatic conversion defined from **double** to **byte**.
- It is still possible to obtain a conversion between incompatible types. ***Casting*** *should be done,*
- *Casting* performs an **explicit conversion** between incompatible types.

Type Conversion and Casting(cntd..)

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.
- Widening conversions works between numeric types. But there are no automatic conversions from the numeric types to **char** or **boolean**

Type Conversion and Casting(cntd..)

Casting Incompatible Types

- What if we want to assign an **int** value to a **byte** variable?
- This kind of conversion is sometimes called a ***narrowing conversion***, since we are explicitly making the value narrower so that it will fit into the target type.
- To create a conversion between two incompatible types, you must use a cast.
- A *cast* is simply an explicit type conversion.
- **General form:**
- **(*target-type*) *value***

- **target-type* specifies the desired type to convert the specified value to.

- If the integer's value is larger than the range of a **byte**, it will be reduced to modulo (the remainder of an integer division) **byte**'s range.

Type Conversion and Casting(cntd.,)

Example 1:

int a;

byte b;

b = (byte) a;

Example 2:

int i = 257 % 127=3

byte b;

b = (byte) i; 00000000 00000000 00000001 00000001

- A different type of conversion will occur when a floating-point value is assigned to an integer type: ***truncation*.**
- Example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.

Cntd..,


```
//Program to 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);
}
}
```

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
```

Cntd..,

Automatic Type Promotion in Expressions

```
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 sub expression **a * b** is performed using integers—not bytes.

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

Cntd..,

The Type Promotion Rules

- Java defines several *type promotion* rules that apply to expressions.
- All **byte**, **short**, and **char** values are promoted to **int**.
- 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**.

Cntd..,

Program:

```
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);
}
}
```

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 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[ ];
- For example, the following declares an array named month_days with the type “array of int”:
- Int month_days[];
- 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.
- array-var = new type[size];

One-Dimensional Arrays cntd

- **Accessing Array elements**
- 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]);

One-Dimensional Arrays cntd

- **Accessing Array elements**
- 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 te value stored at index 3.
- System.out.println(month_days[3]);

One-Dimensional Arrays cntd

DSCE
Dept. of Information Science & Engg

Write a Program to illustrate one dimensional Array

```
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];   // int month_days[] = new int[12];
// int 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.");
```

Output:

April has 30 Days

One-Dimensional Arrays cntd


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

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.
- For example, the following declares a two dimensional array variable called **twoD**.
- Int twoD[][] = new int[4][5];

// Demonstrate a two-dimensional array

Multidimensional Arrays

DSCE
Dept. of Information Science & Engg

```
// 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
```

CHAPTER 2: OPERATORS

Arithmetic Operators

The operands of the arithmetic operators must be of a numeric type.

The following table lists the arithmetic operators:

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

Arithmetic Operators


```
// Demonstrate the basic arithmetic operators.
Class BasicMath {
public static void main(String args[])
 {
// arithmetic using integers
 System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d); System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
```

Cntd..,


```
double da = 1 + 1;
double db = da * 3;
 double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
 System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

Output:

Integer Arithmetic a = 2
b = 6
c = 1
d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

The Modulus Operator

- The modulus operator, **%**, returns the remainder of a division operation.
- It can be applied to floating-point types as well as integer types.

Example:
```
class Modulus{
    public static void main(String args[])
{       int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
}
}
```

output:

x mod 10 = 2

y mod 10 = 2.25

Arithmetic Compound Assignment Operators

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

  Example 1: a = a + 4;

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

  a += 4;

  Example 2: a = a % 2;

- It can be expressed as a %= 2;
- Thus, any statement of the form

  var = var op expression;

- The above can be rewritten as

  var op= expression;

Arithmetic Compound Assignment Operators cntd..

- 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, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

Arithmetic Compound Assignment Operators cntd..

Example:

```
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);
}
}
```

Output:

a = 6
b = 8
c = 3

Increment and Decrement

- The increment operator increases its operand by one. The decrement operator decreases its operand by one.
- For example, this statement :x = x + 1;

  can be rewritten like this by use of the increment operator:

  x++;
- Similarly, this statement :x = x - 1; is equivalent to

  x--;
- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form, the previous value is obtained for use in the expression, and then the operand is modified

Increment and Decrement cntd..

- Example 1: x = 42;

  y = ++x;

  y=43; x=43

- y = ++x; is equivalent to these two statements:

  x = x + 1;

  y = x;

- Example 2: x = 42;

  y = x++;

  y=42; x=43

- y = x++; is equivalent to these two statements:

  y = x;

  x = x + 1;

Increment and Decrement cntd..

```
// Demonstrate ++.
class IncDec {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

The output of this program follows:

a = 2
b = 3
c = 4
d = 1

The Bitwise Operators

- Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte.

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

The Bitwise Operators

- **The Bitwise Logical Operators**
- The bitwise logical operators are &, |, ^, and ~.

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

The Bitwise Operators

1. The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

2. The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

00101010 42

&

00001111 15

00001010 10

The Bitwise Operators

3. The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown

```
 00101010   42
|
00001111    15
 00101111    47
```

4. **The Bitwise XOR**

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
 00101010   42
^
00001111    15
 00100101    37
```

The Bitwise Operators

5. The Left Shift

- The left shift operator, <<< shifts all of the bits in a value to the left a specified number of times. It has this general form:
- value << num
- Here, num specifies the number of positions to left-shift the value in value.
- That is, the << moves all of the bits in the spe
- cified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

The Bitwise Operators

6. The Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is :

  value >> num

- Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.
- The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

Example: int a = 32;

a = a >> 2; // a now contains 8

00100011 35

>> 2

00001000 8

The Bitwise Operators

6. The Unsigned Right Shift, >>>

- If you are shifting something that does not represent a numeric value, you may not want sign extension to take place
- . In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.
- To accomplish this, you will use Java's unsigned, shift-right operator,

>>>, which always shifts zeros into the high-order bit.

Example:

int a = -1;

a = a >>> 24;

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int

>>>24

00000000 00000000 00000000 11111111 255 in binary as an int

The Bitwise Operators

- Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13;

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The Bitwise Operators cntd..

- Assume integer variable A holds 60 and variable B holds 13 then:

```
public class Test
 {
 public static void main(String args[])
{     int a = 60;     /* 60 = 0011 1100 */
       int b = 13;    /* 13 = 0000 1101 */
       int c = 0;
  c = a & b;       /* 12 = 0000 1100 */
       System.out.println("a & b = " + c );
  c = a | b;       /* 61 = 0011 1101 */
   System.out.println("a | b = " + c );
 c = a ^ b;       /* 49 = 0011 0001 */
  System.out.println("a ^ b = " + c );
```

The Bitwise Operators Cntd..

```
 c = ~a;          /*-61 = 1100 0011 */
 System.out.println("~a = " + c );
   c = a << 2;     /* 240 = 1111 0000 */
  System.out.println("a << 2 = " + c );
c = a >> 2;     /* 215 = 1111 */
System.out.println("a >> 2  = " + c );
   c = a >>> 2;     /* 215 = 0000 1111 */
  System.out.println("a >>> 2 = " + c );
 }}
```

Output:

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

Relational Operators..

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

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

Relational Operators..

- 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.**

For **example,**

```
int a = 4;
int b = 1;
boolean c = a < b;
```

Boolean Logical Operators

- The Boolean logical operators shown here operate only on **boolean operands.**

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

Boolean Logical Operators cntd..

- **Example**: Assume Boolean variables A holds true and variable B holds false, then:

```
public class Test
 {
 public static void main(String args[])
{
 boolean a = true;
boolean b = false;
System.out.println("a && b = " + (a&&b));
System.out.println("a || b = " + (a||b) );
  System.out.println("!(a && b) = " + !(a && b));
}
}
```

Output:

```
a && b = false
a || b = true
!(a && b) = true
```

Assignment operators:

- The assignment operator works in
- Java much as it does in any other computer language. It has this general form:

  var = expression;

- **For example**, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

The ? Operator

- It is also known as Conditional operator or the **ternary operator.**
- This operator consists of three operands and is used to evaluate Boolean expressions.
- The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:
- **variable x = (expression) ? value if true : value if false**

The ? Operator

Following is the **Example**:

```
public class Test {
  public static void main(String args[])
{      int a , b;
         a = 10;
  b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " +  b );
    b = (a == 10) ? 20: 30;
  System.out.println( "Value of b is : " + b );
  }}
```

This would produce the following result:

Value of b is : 30**Value of b is : 20**

Operator Precedence

- Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated.
- Certain operators have higher precedence than others;
- for example, the multiplication operator has higher precedence than the addition operator:
- For example, x = 7 + 3 * 2; here x is assigned 13, not 20

Operator Precedence

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

Operator Precedence

- **Using Parentheses**
  - *Parentheses raise the precedence of the operations that are inside them*
  - For example, consider the following expression:

    a >> b + 3
  - That is, this expression can be rewritten using redundant parentheses like this:

    a >> (b + 3)
  - However, if you want to first shift **a right by b positions and then add 3 to that result,** you will need to parenthesize the expression like this:

    (a >> b) + 3
  - Parentheses can sometimes be used to help clarify the meaning of an expression.
  - For example, which of the following expressions is easier to read?

    (i) a | 4 + c >> b & 7

    (ii) (a | (((4 + c) >> b) & 7))

Control Statements

- 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 programcontrol statements can be put into the following categories:
- selection,
- iteration, and
- jump.

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.
- Here is the general form of the **if statement:**

  if (*condition) statement1;*

  else *statement2;*

Example: consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Java's Selection Statements

2. 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.
- **Here is an example:**

```
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)
```

Java's Selection Statements

3. 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;