Introduction to Java

- History: Java is based on the two lang. C and C++.
- C lang.(by Dennis Ritchie) is the first high level programming lang. Structured/Procedure oriented lang.
- C++ lang.(by Bjarne Stroustrup) is the extension of C(‘C’ + ‘OOP’).C++ is partially Object oriented lang.

- Java is developed by James Gosling, patrick naughton, chris warth, Ed frank and Mike sheriden at Sun Microsystems Inc. in 1991. Its name was “Oak” at that time. Later in 1995 it is renamed as Java.
- The primary motivation of java was platform independence lang. that can be used to create software (embedded s/w) for consumer electronics devices.

- Later it is observed that java can be used for internet also(due to platform-independency). Today most browsers are integrated with java runtime environment(JRE).The key point of java for internet are
- Applet(client-side), Servlets(server-side)
- Security(control access within Java execution environment only, not to other part of Comp.)
- Portability(Same applet code work on all system.)

- Bytecode: The security and portability feature of java comes from bytecodes. The java compiler(javac) generate output for a virtual machine(JVM) not for the real machine / real platform. The java interpreter (java) executes the bytecodes and produce output.
- JVM differs platform to platform but all JVM understand same bytecode. Thus as long as JVM is installed all java prog. will run irrespective of h/w platform.

- Java Features

a) Simple
b) Secure
c) Portable
d) Object oriented
e) Robust
f) Multithreaded
g) Architecture neutral
h) Interpreted
i) High performance
j) Distributed
k) Dynamic

- The latest version of java is Java SE 8(internally 1.8).
- J2SE:Java 2 standard edition(JRE, Packages)
- J2EE:Java 2 enterprise edition (J2SE+ JSP, Beans, Framework, API etc)
- J2ME:Java 2 micro edition(for mobile and portable devices)

- Some basic Java tools

javac: Compiler for Java

java: The launcher for Java applications.

javadoc: API documentation generator.

apt: Annotation processing tool.

appletviewer: Run and debug applets

jar: Create and manage Java Archive (JAR) files.

jdb: The Java Debugger.

javah: C header and stub generator.

javap: Class file disassembler

- OOP and Java

OOP and Java

- The two approaches of programming

a) Structured/process-oriented: Like in C. Follow top down approach.

b) Object Oriented: Like in java. Follow bottom up approach.

OOP

- The principle of OOP are

a) Abstraction: Hiding complexity.(object in java, Car as a object not as a collection of various parts like engine, break, gear etc).

b) Encapsulation: Binding code and data together and keep safe from outside interference.(A class in java).

c) Inheritance: Reuse of existing class/code.(More feature are added in subclass then base class).

d) Polymorphism: Different behavior for different inputs.(One interface shared by multiple methods).

A simple java Prog.

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

- Save the file with name Example.java(as the class name is Example which contains the main method.)
- Compile the program with

  >javac Example.java

  This will create a class file Example.class.(The compiler will create a separate class file for each class. A class file is nothing but the bytecodes for JVM).
- Execute the program with

  >java Example

  This will run the code and give the output message.

  This is a simple Java program.

- If the program contain more then one class the file name will be the class name which contain main() method.
- By doing this we will get the class file name (Example.class) same as the class name it contain.(class Example).
- When we execute ‘java Example’, then we are actually specifying the class name that we want to execute. So the interpreter will search for a file classname.class(Example.class)

- Comments: // single line comment

```
/*   Multiline
     comment  */
/** documentation
     comments  for javadoc */
```

- ‘class Example’ declares a class with name Example.
- ‘public static void main(String args[])’ declares main method.
- public is access specifier, which control the visibility of class member. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared. In this case, main( ) must be declared as public, since it must be called by code outside of its class when the program is started.

- The keyword static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java interpreter before any objects are made.
- The keyword void simply tells the compiler that main( ) does not return a value
- Java is case-sensitive. Thus, Main is different from main.(with Main() the program will compile but on execution give error main() not found).

- In main( ), there is only one parameter. String args[ ] declares a parameter named args, which is an array of instances of the class String. (*Arrays are collections of similar objects.)*

- System.out.println("This is a simple Java program.");

  This line outputs the string “This is a simple Java program.” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. System is a predefined class that provides access to the system, and out is the output stream that is connected to the console. (System class, out object and println() method.)

- Question
- What if the program contain more then two main methods.(Error/compile, how to execute that program).
- Is it necessary to keep the file name same as class name. If no then how we will execute the program.

```
class A{
 public static void main(String args[]) {
  System.out.println("Hi Main A");
 }
}
class B{
 public static void main(String args[]) {
  System.out.println("Hi Main B");
 }
}
```

- Save with file name ex1.java, compile it. It will generate two files A.class and B.class.

```
class Example2 {
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);
}
}
```

Data types, Variables and Array

Java Is a Strongly Typed Language

- First, every variable has a type, every expression has a type, and every type is strictly defined.
- Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors

- *For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.*

The Simple Types

- Simple types represents single value
- Java defines eight simple (or elemental) types of data:
- These can be put in four groups:

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

Note: in C and C++ allow the size of an integer to vary based upon execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform.

1. Integers

- Java defines four integer types :
- All of these are signed, positive and negative values. Java does not support unsigned. Many other
- The width and ranges of these integer types vary widely, as shown in this table:

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

- For example:
  - byte a, b;
  - Long p,q,r;
  - Short x,y,z;

Example

- 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.

2. 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.

	Name	Width	Approximate Range
1.	**double**	64	4.9e–324 to 1.8e+308
2.	**float**	32	1.4e−045 to 3.4e+038

- 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);
    }
}
```

3.Character

- Used to store characters
- **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integer type that is 8 bits wide. in Java **char** is a 16-bit type.
- The range of a **char** is 0 to 65,536.
- 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

Another example

- Even though **char**s are not integers, in many cases you can operate on them as if they were integers. This allows you to add two characters together, or to increment the value of a character variable.

- // 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
```

4. Boolean Type

- for logical values.
- It can have only one of two possible values, **true** or **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);
        // a boolean value can control the if statement
        if(b)
                System.out.println("This is executed.");

        b = false;
        if(b)
                System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

Continue…

- The output generated by this program is shown here:

  b is false

  b is true

  This is executed.

  10 > 9 is true

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”

(1) (2) (3) (4)

1. specifies an integer,
2. specifies a floating-point value.
3. specifies a character constant,
4. specifies a string.

Integer Literals.

- Any whole number value is an integer literal. Examples are 1, 2, 3, and 42.
- Octal values are denoted in Java by a leading zero as 07,06 but 09 is not.
- a hexadecimal constant with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f)* are substituted for 10 through 15.

Floating-Point Literals

- They can be expressed in either *standard* or *scientific* notation.
- *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation
- Scientific notation consists of a floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.

  Examples include 6.022E23, 314159E–05

- Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant.

Boolean Literals

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

Character Literals

- Characters in Java are indices into the Unicode character set.
- A literal character is represented inside a pair of single quotes.

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

String Literals

- String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.

  “Hello World”

  “two\nlines”

  “\”This is in quotes\””

- The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals.

Operator and Control statements

Operators in JAVA

- Arithmetic Operators

  + Addition
  – Subtraction (also unary minus)
  * Multiplication
  / Division
  % Modulus
  ++ Increment (pre and post)
  += Addition assignment (a+=b ==> a=a+b)
  –= Subtraction assignment
  *= Multiplication assignment
  /= Division assignment
  %= Modulus assignment
  – – Decrement (pre and post)

- Arithmetic operators cannot use on Boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.
- The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the % can only be applied to integer types.)

  y = ++x; ➔ x=x+1 and y=x;

  y = x++; ➔ y=x and x=x+1;

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

~	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		

- Java uses *two's complement to represent integers.(all signed number except char in JAVA)*
- The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. v*alue << num*
- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. *value >> num*
- When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension and serves* to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

11111000	–8	0001000	8	11111111	-1
>>1		>>1		>>1/2/3….	
11111100	–4	0000100	4	11111111	-1

- The >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value.
- The Java's unsigned, shift-right operator, **>>>,** always shifts zeros into the high-order bit. This is known as an *unsigned shift.*

```
int a = -1;
a = a >>> 24;
```

11111111 11111111 11111111 11111111 –1 in binary

>>>24

00000000 00000000 00000000 11111111 255 in binary

- Relational Operators

== Equal to

!= Not equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

- The result produced by a relational operator is a Boolean value.

- Boolean Logical Operators

&	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

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

expression1 ? expression2 : expression3

max = a>b ? a:b;

```
if(a>b)
    max=a;
else
    max=b;
```

Operator precedence

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

a = 0011	3
b = 0110	6
a|b = 0111	7
a&b = 0010	2
a^b = 0101	5
~a&b|a&~b = 0101	5
~a = 1100	12

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

Control statements

- If else:

  if (*condition) statement1;*

  else *statement2;*

- The *condition is any expression that returns a boolean value.* The else clause is optional.
- Nested if:

```
if(i == 10) {
  if(j < 20) a = b;  //here i==10 is true and this if is
  else a = c;        // associated with this else
}
else a = d;
```

- **if-else-if Ladder**

```
if(condition)
      statement;
else if(condition)
      statement;
else if(condition)
      statement;
      ...
 else
    statement;
```

- Switch case statement:

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

- The *expression must be of type byte, short, int, or char; each of the values specified* in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.
- The default statement is optional. If no case matches and no default is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence. This has the effect of “jumping out” of the switch.

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

- **Loops**: entry control loops and exit control loops

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

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

```
 int i = 100, j = 200;
// find midpoint between i and j
while(++i < --j) ; // no body in this loop, semicolon end loop
 System.out.println("Midpoint is " + i);
```

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

```
int n = 10;
do {
    System.out.println("tick " + n);
    n--;
    } while(n > 0);

//more efficient version of same code.
do {
    System.out.println("tick " + n);
   } while(--n > 0);
```

- The do-while loop is especially useful when we process a menu selection, because we want the body of a menu loop to execute at least once to show the menu.

- for loop:
  for(*initialization; condition; iteration) {*
  // body
  }

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

```
//Declaring Loop Control Variables Inside the for Loop
for(int n=10; n>0; n--)
System.out.println("tick " + n);
```

```
// Test for primes.
class FindPrime {
public static void main(String args[]) {
int num;
boolean isPrime = true;
num = 14;
for(int i=2; i <= num/2; i++) {
if((num % i) == 0) {
isPrime = false;
break;
}
}
if(isPrime) System.out.println("Prime");
else System.out.println("Not Prime");
}
}
```

```
for(i=0; i<10; i++) {
  for(j=i; j<10; j++)
   System.out.print(".");
   System.out.println();
```

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

```
class BreakLoop2 {
   public static void main(String args[]) {
   int i = 0;
   while(i < 100) {
    if(i == 10) break; // terminate loop if i is 10
    System.out.println("i: " + i);
     i++;
    }
    System.out.println("Loop complete.");
  }
}
```

```
// 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
```

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

- The output from this program is:

  Before the return.

Math class
Array(basic,array of Obj,
Pass array to method/return array)

- The Math class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general purpose methods. Math defines two double constants: E (approximately 2.72) and PI (approximately 3.14).

- **static double sin(double *arg)*** *Returns the sine of the angle specified* by *arg in radians.*
- **static double cos(double *arg)*** *Returns the cosine of the angle specified* by *arg in radians.*
- **static double tan(double *arg)*** *Returns the tangent of the angle* specified by *arg in radians.*
- **static double asin(double *arg)*** *Returns the angle whose sine is specified* by *arg.*
- **static double acos(double *arg)*** *Returns the angle whose cosine is* specified by *arg.*
- **static double atan(double *arg)*** *Returns the angle whose tangent is* specified by *arg.*
- **static double atan2(double *x, double y)*** *Returns the angle whose tangent is x/y.*

- **static double exp(double *arg)** *Returns e to the arg.*
- **static double log(double *arg)** *Returns the natural logarithm of arg*
- **static double pow(double *y, double x)** *Returns y raised to the x.*
- **static double sqrt(double *arg)** *Returns the square root of arg.*
- **static int abs(int *arg)** *Returns the absolute value of arg.*
- **static long abs(long *arg)** *Returns the absolute value of arg.*
- **static float abs(float *arg)** *Returns the absolute value of arg.*
- **static double abs(double *arg)** *Returns the absolute value of arg.*

- **static double ceil(double *arg)*** *Returns the smallest whole number greater* than or equal to *arg.*
- **static double floor(double *arg)*** *Returns the largest whole number less than* or equal to *arg.*
- **static int max(int *x, int y)*** *Returns the maximum of x and y.*
- **static double max(double *x, double y)*** *Returns the maximum of x and y.*
- **static int min(int *x, int y)*** *Returns the minimum of x and y*
- **static long min(long *x, long y)*** *Returns the minimum of x and y.*
- **static double rint(double *arg)*** *Returns the integer nearest in value to arg.*
- **static int round(float *arg)*** *Returns arg rounded up to the nearest int.*
- **static long round(double *arg)*** *Returns arg rounded up to the nearest long.*

- **static double IEEEremainder(double *dividend, double divisor)***
- **static double random( )**
- **static double toRadians(double *angle)***
- **static double toDegrees(double angle)**
- **IEEEremainder( )** returns the remainder of *dividend/divisor. random( ) returns a* pseudorandom number. This value will be between 0 and 1. Most of the time, we will use the **Random** class when you need to generate random numbers.
- The **toRadians( )** method converts degrees to radians. **toDegrees( )** converts radians to degrees. The last two methods were added by Java 2

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.
- *Arrays in Java work differently than they do in C/C++ languages.*

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

  type var-name[ ];

- 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, we 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];
```

- Once we 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]);
```

- 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];
```

- Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range.
- In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks. For example, the run-time system will check the value of each index into month_days to make sure that it is between 0 and 11 inclusive.
- If we try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), we will cause a run-time error.

- In Java, *multidimensional arrays are actually arrays of arrays. These, as you might* expect, look and act like regular multidimensional arrays

```
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.*

- We can create variable length multi-dimensional array as:

```
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
```

- There is a second form that may be used to declare an array:

  type[ ] var-name;

- Here, the square brackets follow the type specifier, and not the name of the array variable. For example, 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];
```

- This alternative declaration form is included as a convenience, and is also useful when specifying an array as a return type for a method.

- The size of an array—that is, the number of elements that an array can hold—is found in its length instance variable. All arrays have this variable, and it will always hold the size of the array.
- Array of objects can be obtained as

  A a_obj[3]=new A();

  Here a_obj[3] is an array of objects of class A.(a_obj[0], a_obj[1] and a_obj[2])

```
class A {
void display(int []a1)  { System.out.println("a1 is " +a1.length);}
int[] change(int [] tmp) {  for(int i=0;i<tmp.length;i++)
                                    tmp[i]=I;     return (tmp);  }
public static void main(String args[])   {
int a1[] = new int[10];    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};      int a4[]=a2;//Now a4 is same as a2
A a_obj=new A();
a_obj.display(a1);//pass array to methods
a4=a_obj.change(a2);// Pass and return array to method
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
for(int i=0;i<a4.length;i++)    System.out.print(" "+ a4[i]);
  System.out.println("");
for(int i=0;i<a2.length;i++)   System.out.print(" "+ a2[i]);   } }
```

Classes and Objects

- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template for an object, and an object is an instance of a class. Because an* object is an instance of a class, the two words *object and instance used* interchangeably.

General form of class

class *classname {*
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}

A simple class

```
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
```

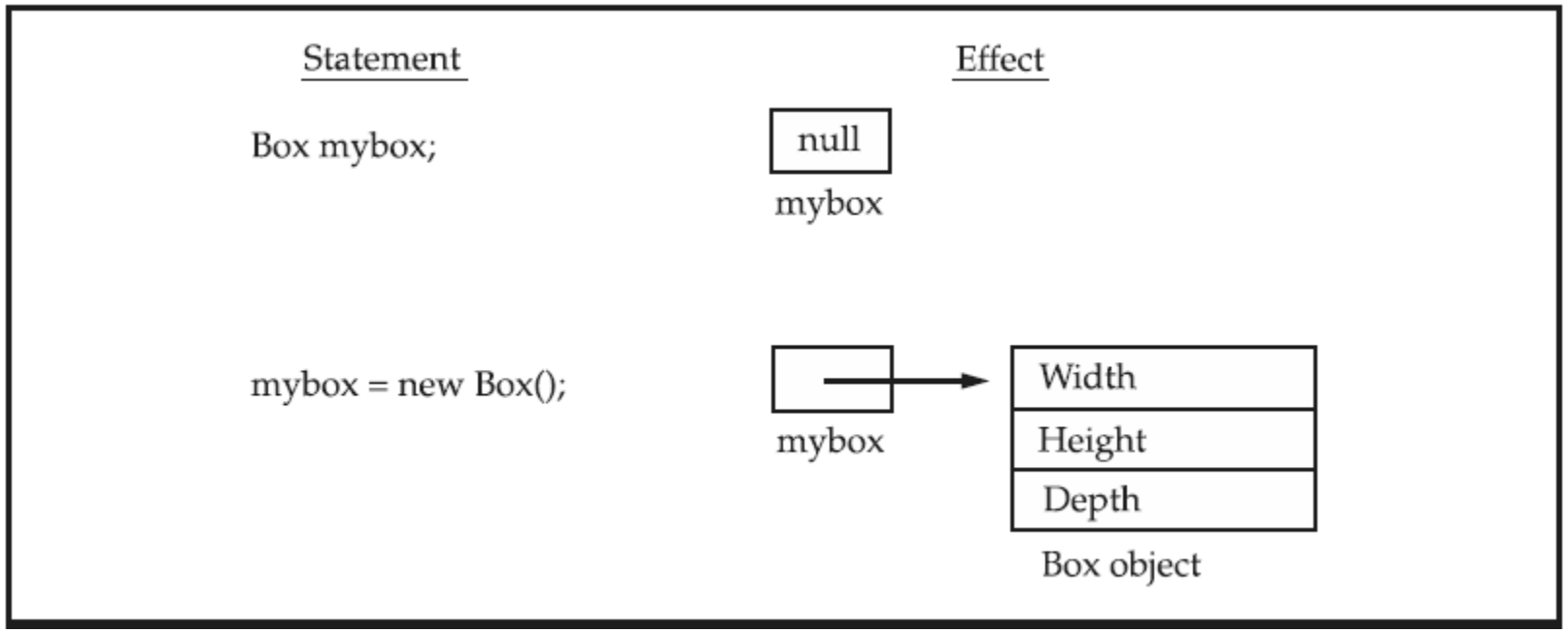
```
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

O/p Volumn is 3000.00

- We can create as many object as needed. Like
- Box mybox1 = new Box();
- Box mybox2 = new Box();

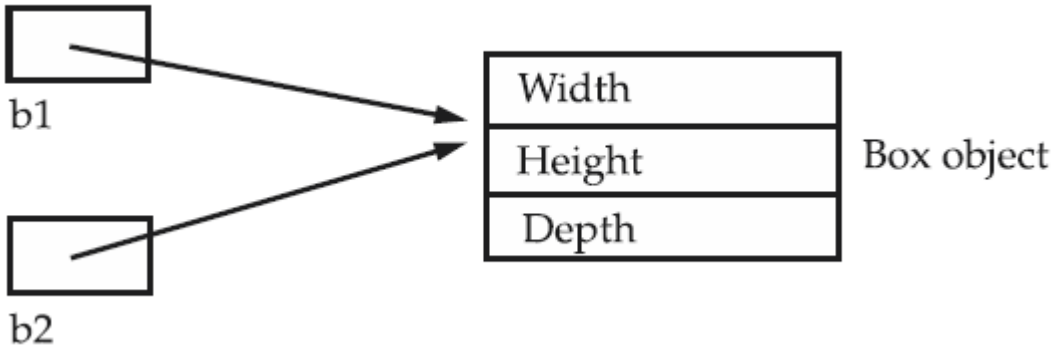
  will create two objects mybox1 and mybox2 each with separate instance values(width, height and depth).

Creating Object



- Assigning Object Reference Variables:
- Object reference variables act differently than an assignment. For example
- Box b1 = new Box();
- Box b2 = b1;

  here b2 is assigned to b1 object, but no separate space is allocated to b2. b2 is a reference to the same memory location as of b1.




```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1 has been set to null, but b2 still points to the original object.**

Introducing Methods

```
class Box {
double width;
double height;
double depth;
// display volume of a box
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

```
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
// display volume of first box
mybox1.volume();
}
}
```

Returning a Value

```
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
}
}
```

Adding a Method That Takes Parameters

```
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
```

```
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Next class

- Constructors
- Parameterized Constructors
- The this Keyword
- Garbage Collection
- The finalize( ) Method

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created. (Although by methods we can do so). Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- A *constructor initializes an object immediately upon creation. It has the same name* as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

- Constructors have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Example with constructor

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

O/P->

- Constructing Box
- Constructing Box
- Volume is 1000.0
- Volume is 1000.0

- When we allocate an object, we use the following form: *class-var = new classname( );*
- This explains why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line: Box mybox1 = new Box();

  new Box( ) is calling the Box( ) constructor. When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- This is why the preceding line of code worked in earlier versions of Box that did not define a constructor. The default constructor automatically initializes all instance variables to zero(default values). Once we define our own constructor, the default constructor is not used.

Parameterized Constructors

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d; }
// compute and return volume
double volume() {
return width * height * depth;
}  }
```

```
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

O/p->

```
Volume is 3000.0
Volume is 162.0
```

The this Keyword

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the *current object.*
- That is, this is always a reference to the object on which the method was invoked. To better understand what this refers to, consider the following version of Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Instance Variable Hiding

- When a local variable has the same name as an instance variable, the local variable *hides the instance variable. So width, height, and* depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If

  they had been, then width would have referred to the formal parameter, hiding the instance variable width.
- In such cases this keyword can be used to resolve any name space collisions that might occur between instance variables and local variables.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

Garbage Collection

- Objects are dynamically allocated by using the new operator in java. The memory management for such dynamic allocation is necessary.
- In C++ language dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for us automatically. The technique that accomplishes this is called *garbage collection.*
- *When no references to an object exist, that* object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.

- Garbage collection only occurs sporadically (irregularly) during the execution of program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations takes varying approaches for garbage collection.
- To run the garbage collection manually use this code.

```
Runtime r = Runtime.getRuntime();
r.gc();// void gc() is function for garbage collector.
```

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then we want to make sure that these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.*
- *By using finalization, we can define specific actions that will occur* when an object is just about to be reclaimed by the garbage collector.

- The Java run time calls finalize method whenever it is about to recycle an object of that class. Inside the finalize( ) method we will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.
- The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- protected is a specifier that prevents access to finalize( ) by code defined outside its class.
- finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that we cannot know when finalize( ) will be executed. Therefore, our program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize( ) for normal program operation.
- *C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The finalize( ) method only approximates the function of a destructor. The need for destructor functions is minimal because of Java's garbage collection subsystem.*

A Stack Class

```
// This class defines an integer stack that can hold 10 values.
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {  tos = -1; }
// Push an item onto the stack
void push(int item) {
if(tos==9)    System.out.println("Stack is full.");
else  stck[++tos] = item;  }
// Pop an item from the stack
int pop()   {
   if(tos < 0) {  System.out.println("Stack underflow.");  return 0;  }
   else
     return stck[tos--];
  }  }
```

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
}
}
```

A few points about stack class

- The implementation details are irreverent as the access to class members is done by method push() and pop(). So the stack is implemented by array or by link list does not affect the user of class who is accessing it by methods.
- It is possible for the array that holds the stack, ‘stck[]’ or ‘tos’ member to be altered by code outside of the Stack class. This leaves Stack open to misuse or mischief.(Should be declared private).

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

A Closer Look at Methods and Classes

- Overloading Methods
- Overloading Constructors
- Objects as Parameters
- Argument Passing
- Returning Objects

Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded, and the process is referred to as method overloading.*
- Method overloading is one of the ways that Java implements polymorphism.

- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

O/p->

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Automatic type conversion and Overloading

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution.

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2
```

- This version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. But, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Therefore, after test(int) is not found, Java elevates i to double and then calls test(double).
- Java will employ its automatic type conversions only if no exact match is found.

  [*Complied time polymorphism is implemented by method overloading. Run-time polymorphism is implemented by Method overriding.(dynamic method dispatch: a call to an overridden method is resolved at run time, rather than compile time.)]

- Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name.
- For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing and the underlying concept of each function is the same.
- This situation does not occur in Java, because each absolute value method can use the same name. In java abs() method is overloaded by Java's Math class to handle all numeric types. Java determines which version of abs( ) to call based upon the type of argument.

Overloading Constructors

```
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d; }
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box  }
// constructor used when cube is created
Box(double len) {
width = height = depth = len; }
// compute and return volume
double volume() {
return width * height * depth; }
 }
```

```
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
vol = mybox1.volume(); // get volume of first box
System.out.println("Volume of mybox1 is " + vol);
vol = mybox2.volume();  // get volume of second box
System.out.println("Volume of mybox2 is " + vol);
vol = mycube.volume(); // get volume of cube
System.out.println("Volume of mycube is " + vol);
}  }
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Objects as Parameters

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
```

```
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

- One of the most common uses of object parameters involves constructors. If we want to construct a new object which is initially as same as some existing object, then we must define a constructor that takes an object of its class as a parameter.

```
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) {   // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
double volume() {
return width * height * depth; }
 }
```

```
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box myclone = new Box(mybox1);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
}
}
```

O/p->

```
Volume of mybox1 is 3000
Volume of myclone is 3000
```

Argument Passing

- In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. (call by value).
- Objects are passed by reference. Changes to the object inside the method *do affect the object used as* an argument.(call by reference).

Returning Objects

- A method can return any type of data, including class types that you create.

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {  //return type is ‘class instance’ of class Test
Test temp = new Test(a+10);
return temp;
}
}
```

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "
+ ob2.a);
}
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

A Closer Look at Methods and Classes-II

- Recursion
- Access Control
- static keyword
- final keyword
- Arrays
- Nested and Inner Classes
- String Class and command line args
- Variable number of args

Recursion

- Java supports *recursion. Recursion is the process of defining something in terms of* itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive.*

```
// A simple example of recursion.
class Factorial {
// this is a recursive function
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
```

```
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Access Control

- Encapsulation provides another important attribute: *access control. Through* encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.
- Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.(protected and private protected)

```
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

static keyword

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. We can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

- Methods declared as static have several restrictions:

1. They can only call other static methods.(without using any object.)

   *[With ‘object.methodname’ they can call non- static method, like we do in main() which is static. But in general static method are created to access them without using any object.]

2. They must only access static data.
3. They cannot refer to ‘this’ or ‘super’ in any way.

   (The keyword ‘this’ is used to refer to current instance/object while the keyword ‘super’ is related to inheritance.(refer to super/base class.))

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void math(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {  //the static block. It will be exe. automatically and will exe. only
    once.
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
math(42);
} }
```

O/p-> Static block initialized.
x = 42
a = 3
b = 12

- If we need to do computation in order to initialize our static variables, we can declare a static block which gets executed exactly once, when the class is first loaded.
- As soon as the UseStatic class is loaded, all of the static statement runs. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

- *It is illegal to refer to any instance variables inside of a static method.*
- if we wish to call a static method from outside its class, we can do so using the following general form: *classname.method( )*
- Here, *classname is the name of the class in which the static method is declared.* A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output of this program:

a = 42

b = 99

```
public class Number_Objects
{
static int count=0;
Number_Objects()
{
count++;
}
public static void main(String[] args)
{
Number_Objects obj1 = new Number_Objects();
Number_Objects obj2 = new Number_Objects();
Number_Objects obj3 = new Number_Objects();
Number_Objects obj4 = new Number_Objects();
System.out.println("Number of objects
created:"+count);
}
}
```

Final keyword

- A variable can be declared as final. Doing so prevents its contents from being modified. This means that we must initialize a final variable when it is declared. (In this usage, final is similar to const in C/C++.)

  final int FILE_NEW = 1;

  final int FILE_OPEN = 2;

  final int FILE_SAVE = 3;
- Variables declared as final do not occupy memory on a per-instance basis. (final variable uses single memory location which is shared by all instance/objects.). So a final variable is essentially a constant.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. (related with inheritance.)

Arrays

- Array are implemented as objects. Because of this, there is a special array attribute that we can take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its length instance variable. All arrays have this variable, and it will always hold the size of the array.

```
// This program demonstrates the length array member.
class Length {
public static void main(String args[]) {
int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
}
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as *nested classes.*
- The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

  [Inner class can access all members of outer class but outer class can't access member of inner class.]

- There are two types of nested classes:
  static and non-static.
- *A static nested class is one* which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- The most important type of nested class is the *inner class. An inner class is a* non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
} }  }
class InnerClassDemo {
public static void main(String args[]) {                    //Output :
Outer outer = new Outer();                                  //display: outer_x = 100
outer.test();
}  }
```

- In the program, an inner class named Inner is defined within the scope of class Outer. Therefore, any code in class Inner can directly access the variable outer_x. An instance method named display( ) is defined inside Inner. This method displays outer_x on the standard output stream. The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method. That method creates an instance of class Inner and the display( ) method is called.

- class Inner is known only within the scope of class Outer. The Java compiler generates an error message if any code outside of class Outer attempts to instantiate class Inner. Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope.
- An inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

*Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

```
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " + outer_x);
} }
void showy() {
System.out.println(y); // error, y not known here!
} }
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
} }
```

The String Class

- Every string you create is actually an object of type String. Even string constants are actually String objects. For example, in the statement

  System.out.println("This is a String, too");

  the string “This is a String, too” is a String constant. Fortunately, Java handles String constants in the same way that other computer languages handle “normal” strings.

- The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

1. If you need to change a string, you can always create a new one that contains the modifications.
2. Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

Some methods in string class

- s1. charAt(n*);*
- *s2=s1.toLowerCase();*
- *s2=s1.toUpperCase();*
- *s2=s1.replace('x','y');*
- s2=s1.trim();
- s1.equals(s2);
- s1.equalsIgnoreCase(s2);
- s1.length();
- s1.compareTo(s2);
- s1.concat(s2);
- s1.substring(n);
- s1.substring(n,m);
- s1.indexOf('x');

String and command line args

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Execute the code by --->>java CommandLine this is a test 100 -1

output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Variable number of arguments

- Beginning with JDK 5,(v1.5), java has included a feature to take variable number of arguments.
- A sample method is

  void vaTest(int …v);

  which can take any number of integer type arguments. The method can be overloaded for other types as well. Here v is implicitly declared as an array without specifying its size.

```
class A
{
  static void vatest(int ...v)
  {
   System.out.print("Number of args:" + v.length + " contents: ");
   for(int x : v)
    System.out.print(x+" ");

    System.out.println();
  }
   public static void main(String args[])
   {
      vatest(10);
      vatest(1,2,3);
      vatest();
  }
 }
```