Unit-1

History of java:-

- Java was developed by
  - ✓ *James Gosling,*
  - ✓ *Patrik Naughton,*
  - ✓ *Chris Warth,*
  - ✓ *Ed Frank, and*
  - ✓ *Mike Sheridan*

  at Sun Microsystems in 1991.

- It was started in 1991 as a project named “Green”, to develop a platform-independent language

- Java was originally called Oak, because of a tree outside of the window of James Gosling’s office at Sun, and was later renamed to Java when it was discovered a language named Oak already exists.

.

- The name “Java” was chosen because the creators of the language often discussed ideas for the language at a local coffee shop.

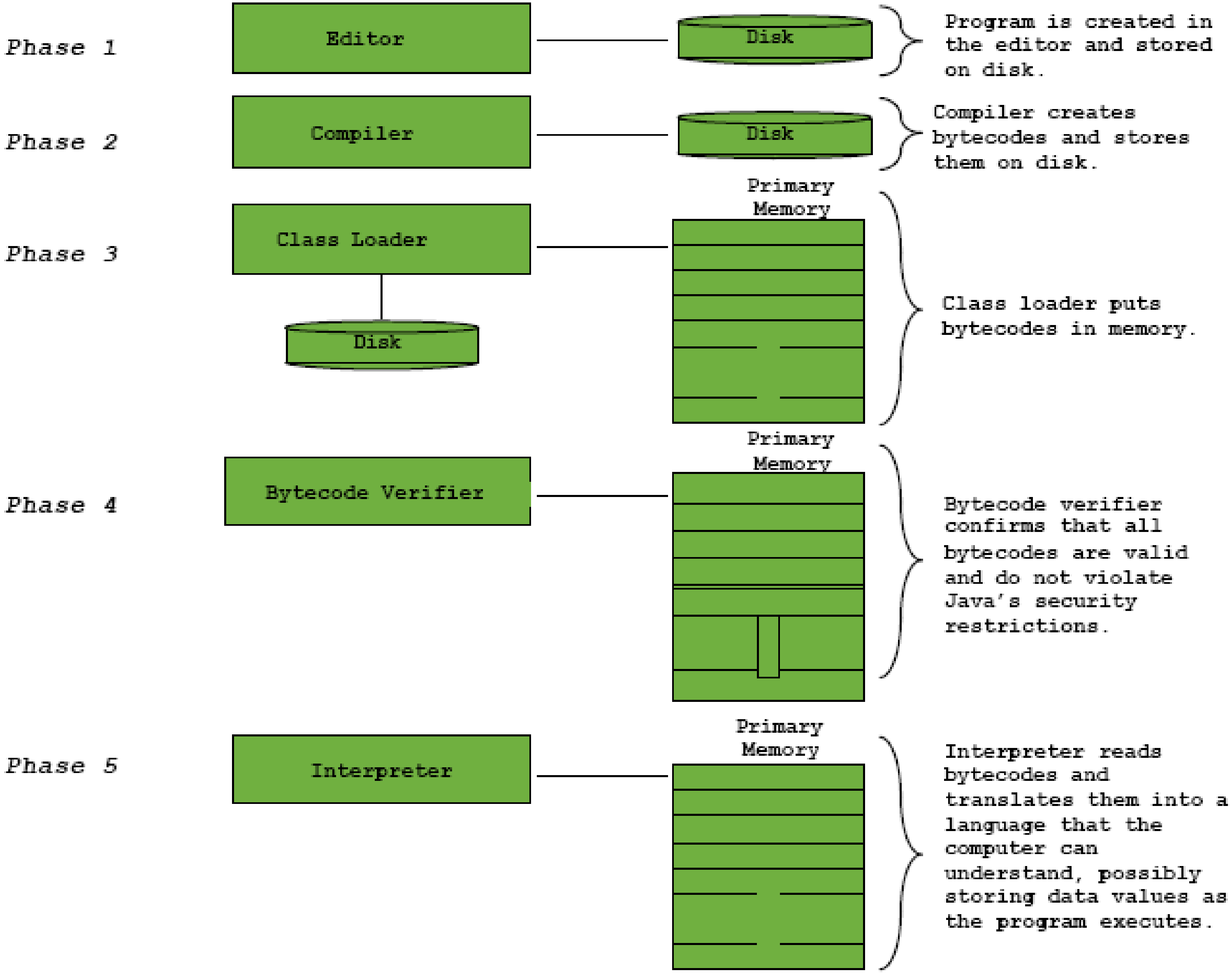
What makes Java unique?

- Java runs inside a piece of software known as a ***Java Virtual Machine***.

- In many languages, such as C and C++, the *source code* (.c and .cpp files) is portable. What makes Java unique is that Java’s *executable code* is also portable (.class files).

- This means that the same application can be run without modification on any system that has a Java Virtual Machine without being recompiled.

- High-level languages such as c and c++ need a different compiler for each type of computer and for each operating system.



Phase 1

Editor

Disk

Program is created in the editor and stored on disk.

Phase 2

Compiler

Disk

Compiler creates bytecodes and stores them on disk.

Phase 3

Class Loader

Disk

Primary Memory

Class loader puts bytecodes in memory.

Phase 4

Bytecode Verifier

Primary Memory

Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.

Phase 5

Interpreter

Primary Memory

Interpreter reads bytecodes and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Java Program Development and Execution Steps

Java Terminology

- **Java Virtual Machine** (JVM)

- **Java Runtime Environment** (JRE) – A runtime environment which includes Java Virtual Machine, and provides all class libraries and other facilities necessary to execute Java programs. *This is the software on your computer that actually runs Java programs.*

Note:

These two terms (JVM and JRE) are often used interchangeably, but it should be noted that they are not technically the same thing.

Contd..

- **Java Development Kit** (JDK) – The basic tools necessary to compile, package Java programs (javac, jar, respectively). *The JDK includes a complete JRE.*

Java Platforms

There are three main platforms for Java:

- **Java SE** (short for Standard Edition) – runs on desktops and laptops.

- **Java ME** (short for Micro Edition) – runs on mobile devices such as cell phones.

- **Java EE** (short for Enterprise Edition) – runs on servers.

Java Versions

- Sun has a history of choosing poor version numbering schemes, and for renumbering a version after it has been publicly released.

Java Version History

- **Java 1.0 (JDK 1.0) – Released in January 1996**
- **Java 1.1 (JDK 1.1) – Released in February 1997**
- **Java 1.2 (JDK 1.2) / J2SE 1.2 (J2SE SDK 1.2) – Released in December 1998**
  - **Development Kit initially named JDK 1.2 but renamed three days after release to J2SE SDK 1.2 (short for Java 2 Standard Edition)**
- **Java 2 Micro Edition (J2ME) and Java 2 Enterprise Edition (J2EE) also released in December 1998.**

Contd..

- Java 2 Standard Edition 1.3 (J2SE SDK 1.3) – Released in May 2000.

- Java 2 Standard Edition 1.4 (J2SE SDK 1.4) – Released in February 2002.

- Java 2 Standard Edition 5.0 (J2SE SDK 5.0) – Released in September 2004.
  - Originally named 1.5 – many official documents still refer to this as version 1.5.

- Java SE 6 (Java SE SDK) – Released in December 2006.
  - J2SE SDK renamed back to Java SE SDK

Java Development Kit (JDK)

- Comes with all tools necessary to compile and run Java programs, including a complete Java Runtime Environment (JRE) (located in the **`jdk/jre`** directory).

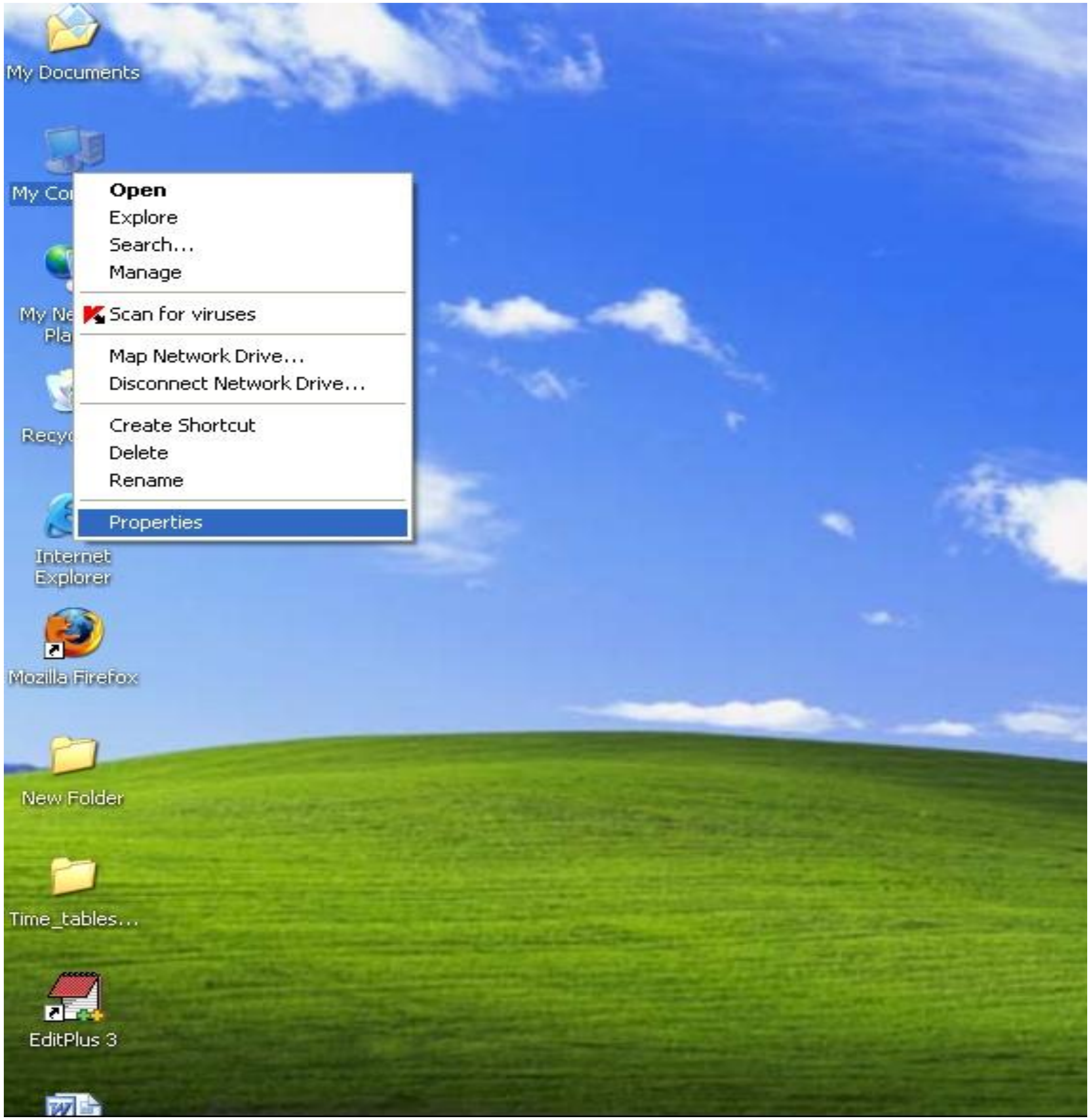
- Get the most recent version from: http://java.sun.com/javase/downloads/index.jsp

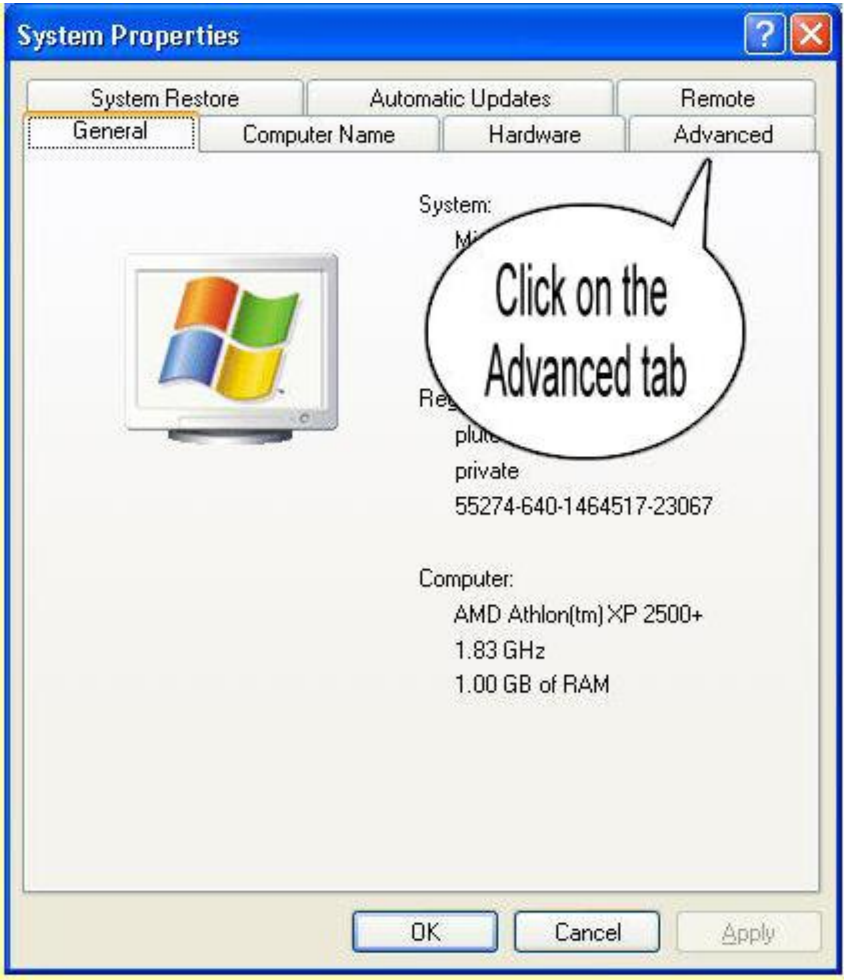
classpath and path

- The PATH is an environment variable that tells the command processor where to look for javac.exe and java.exe

- The CLASSPATH is an environment variable that tells where to look for class files

Setting on windows xp





Click on the Advanced tab

System:

Mi

Reg

plut

private

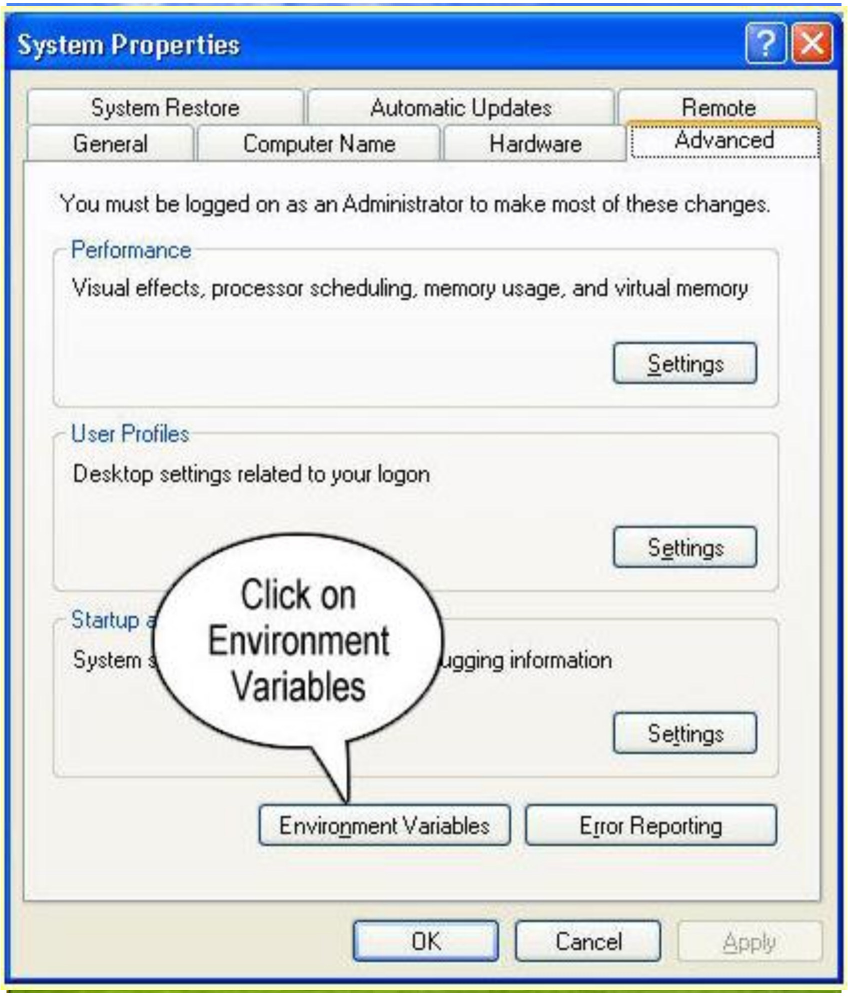
55274-640-1464517-23067

Computer:

AMD Athlon(tm) XP 2500+

1.83 GHz

1.00 GB of RAM



System Properties

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

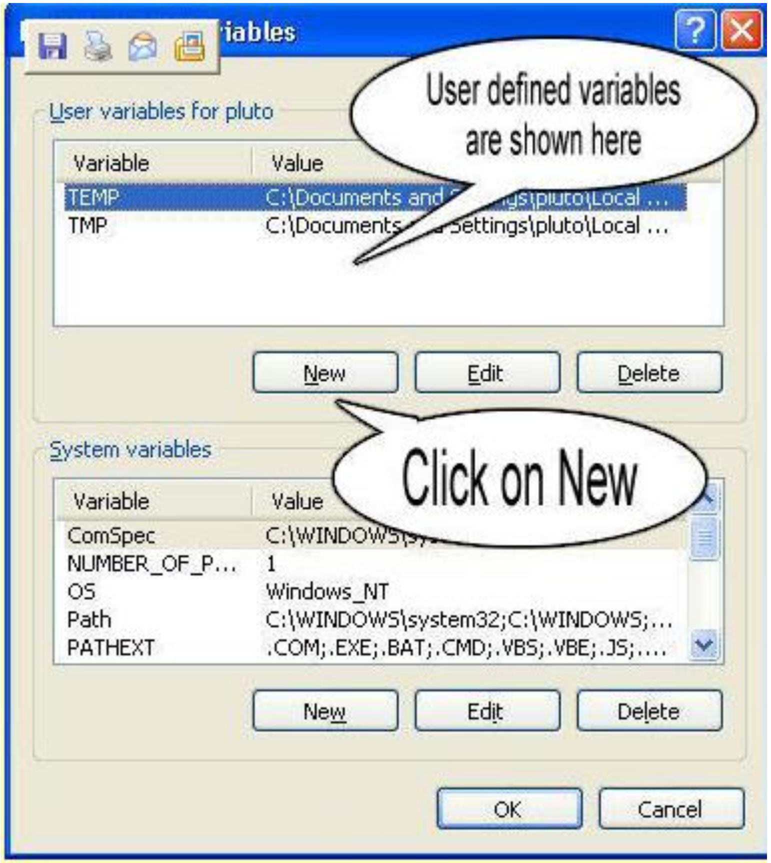
User Profiles

Desktop settings related to your logon

Startup a

System s ugging information

Click on Environment Variables

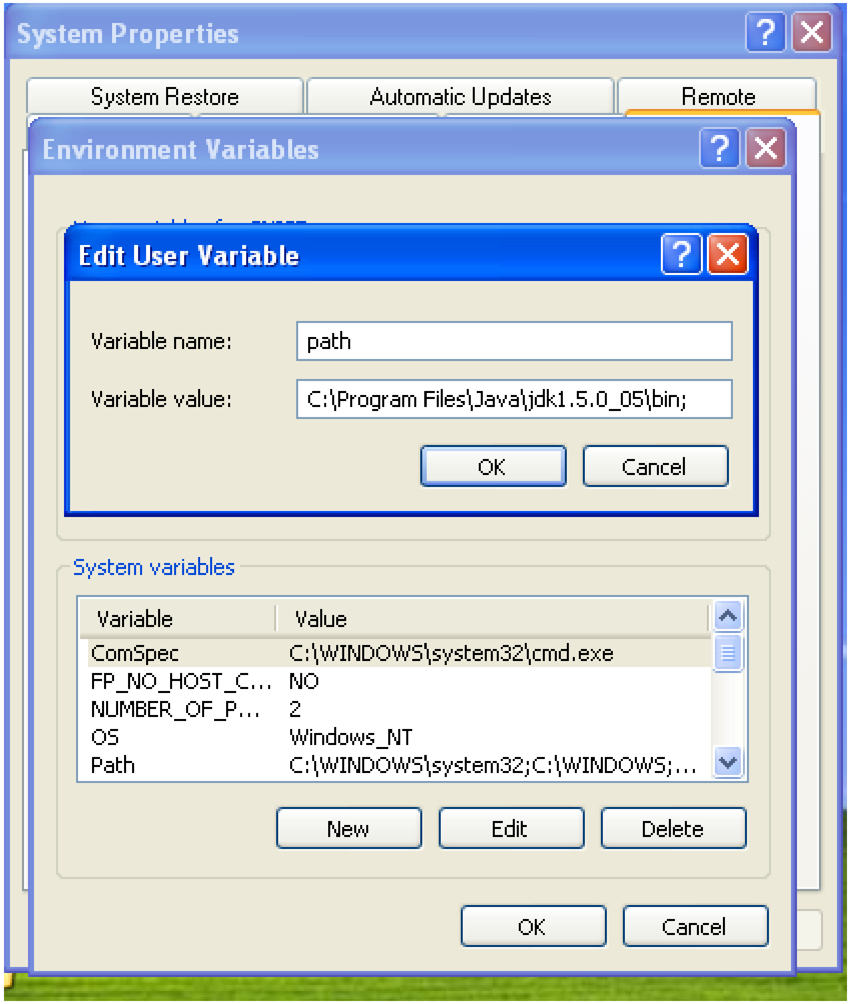


Variable	Value
TEMP	C:\Documents and S…ngs\pluto\Local …
TMP	C:\Documents …d Settings\pluto\Local …

Variable	Value
ComSpec	C:\WINDOWS\sy…
NUMBER_OF_P…	1
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;…
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;….

User defined variables are shown here

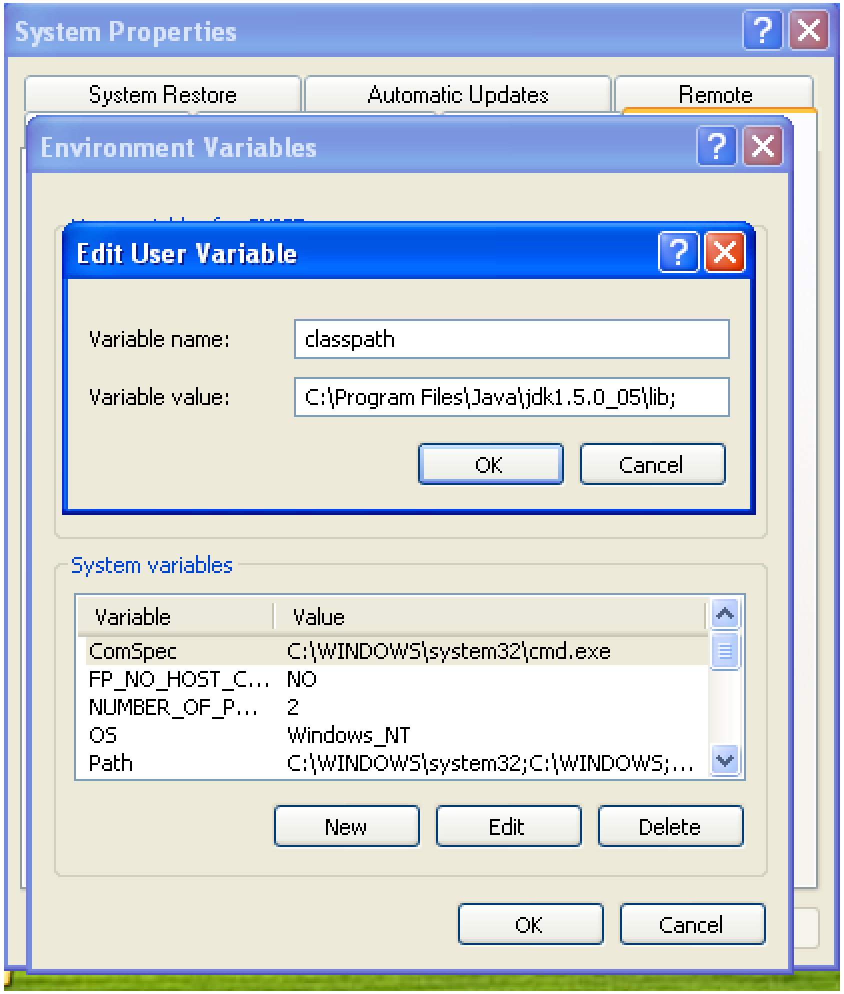
Click on New



Variable name: path

Variable value: C:\Program Files\Java\jdk1.5.0_05\bin;

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;...



Variable name: classpath

Variable value: C:\Program Files\Java\jdk1.5.0_05\lib;

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	2
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;...

Buzz Words (SORSAHMDD)

Simple

- Similar to C/C++ in syntax
- Java eliminates complexities of
  - No operator overloading.
  - No direct pointer manipulation or pointer arithmetic.
  - No multiple inheritance.
  - No malloc() and free() – handles memory automatically.
  - Garbage Collector.
- Lots more things which make Java more attractive.

Object-Oriented

- Fundamentally based on OOP
  - Uses a formal OOP type system.
  - The object model in Java is simple and easy to understand.
  - Efficient re-use of code.
  - Easy to upgrade the system.
  - Easy to maintain etc.

Robust

- Java is highly reliable because
  - Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error.
  - SecurityManager to check which operations a piece of code is allowed to do.

Secure

- Designed with the intention of being secure, because the program must execute reliably in a variety of systems.
  - Java's robustness features help to make it more secure .

Architecture-Neutral/Portable

- “Write once; run anywhere, any time, forever.”
- The Java Virtual Machine becomes the common denominator.
  - Bytecodes are common across all platforms.

High-Performance

- Java performance IS slower than C becuase
    - Interpreted languages will run slowly.
    - Additional checks in Java which make is secure and robust and network aware etc, all have a influence on performance.
- BUT
    - JIT compilation and HotSpot
        - Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
    - HotSpot optimizes code on the fly based on dynamic execution patterns
        - Can sometimes be even faster than compiled C code!

Multi-Threaded

- Java supports multithreaded programming.
  - Java allows us to write programs that do many things simultaneously.
    - Processes - application level.
    - Threads – within the application.

Dynamic

- Java is designed to be able to adapt to an evolving environment in at least two ways:
  - First, at the language level, by adding new packages of classes to its libraries;
  - Second, at the program level, by being able to load in new classes dynamically as the program runs. load new classes at runtime.

Distributed

- Java is designed for the distributed environment of the Internet.
  - Java has built-in classes to support various levels of network connectivity.
  - These are found in the java.net package.

Data Types, Variables

- Java Is a Strongly Typed Language
  - Every variable has a type, every expression has a type, and every type is strictly defined.
  - 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.
- *For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.*

Data Types

The Simple Types

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.

These can be put in four groups:

- *Integers* -This group includes **byte**, **short**, **int**, and **long**, which represent signed integer 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.

integer types

- Java does not support unsigned, positive-only integers**.**
- All are signed, positive and negative values.

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

- byte
  - The smallest integer type is **byte**.
  - Variables of type **byte** are especially useful while working with a stream of data from a network or file.
  - Byte variables are declared by use of the **byte** keyword.

- Ex:-
-                byte z = 22;
-                int a, b, c;
-                int d = 3, e, f = 5;

Floating-Point Types

- There are two kinds of floating-point types.
- All math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

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

Ex:-

```
double pi = 3.14159;
```

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.
- 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.
- It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more.
- Hence it requires 16 bits.
- The range of a **char** in java is 0 to 65,536.
- There are no negative **char**s.

Booleans

- It can have only one of two possible values, **true** or **false**.
- This is the type returned by all relational operators, such as **a < b**.

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

- In Java, all variables must be declared before they can be used.

type identifier;

type identifier = *value*;

Ex:-

```
double pi = 3.14159;
```

Constants

- Integer constants
  - Decimal values without a fractional component.

3 types

- Base 10 constants (*decimal numbers*)
  - The range of decimal digit is *0 to 9, ex:- 2*
- Base 8 constants (*octal numbers*)
  - The range of octal digit is *0 to 7*
  - Octal values are with a leading zero.
  - Ex: - 012
- Base 16 constants (*hexadecimal numbers*)
  - The range of hexadecimal digit is *0 to 15,* so *A* through *F* (or *a* through *f)* are substituted for 10 through 15.
  - Hexadecimal values are denoted with leading zero-x(0x or 0X).
  - Ex:- 0x12

- Default int constant size is 32 bits
- Long int constants are specified by appending an upper- or lowercase *L* to the constant.

  Ex:- 0xfffffffffffffL, 6893468939L .

Floating-Point constants

- Decimal values with a fractional component.
- Representation
  - *Standard notation , ex:- 2.0, 3.14159*
  - *Scientific notation ex:-* 6.022E23, 314159E–05, 2e+100.

- Floating-point constants in Java default to **double** precision.
- To specify a **float** constant, you must append an *F* or *f* to the constant.
  - Ex:- 3.14f

Boolean Constants

- There are only two logical values that a **boolean** value can have, **true** and **false**.
- Only in lower case.

Character constants

- A character constant is represented inside a pair of single quotes.

- All of the visible *unicode* characters can be directly entered inside the quotes, such as *‘a’, ‘z’,* and *‘@’.*

- For characters that are impossible to enter directly, there are several escape sequences, which allow you to enter the character you need.
- Ex:- ‘\’’ for the single-quote character itself, and **‘\n’** for the newline character.

Character Escape Sequences

Escape Sequence	Description
\’	Single quote
\”	Double quote
\\	Backslash
\n	New line (also known as line feed)
\t	Tab
\b	Backspace

String Constants

- Sequence of characters between a pair of double quotes.
  - Examples of string literals are
    - “Hello World”
    - “two\nlines”
    - “\”This is in quotes\””
- A string constant must begin and end on the same line.
- In java strings can be concatenated by using + operator.
  - Ex:- “wel” + “ come” ;

java program structure

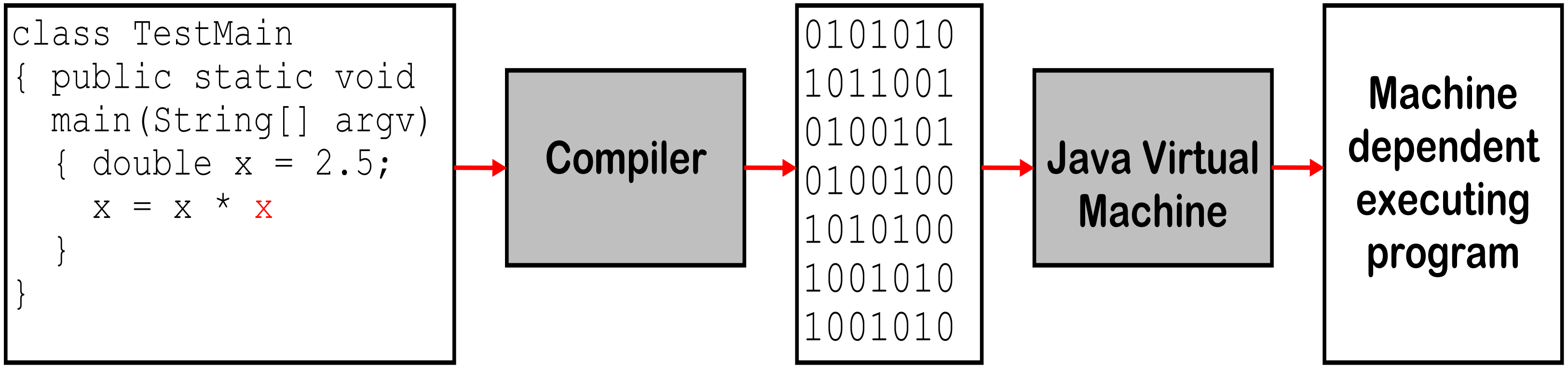
- In Java :
  - A program is made up of one or more *classes.*
  - One class is marked as the special “starting” class.
  - Name of the file should coincide with the name of starting class.
  - When a Java program is launched by the interpreter (JVM), it invokes a static method named “main” in the start class.

Ex:-

```
public class ClassName
  {
      public static void main(String args[])
      {
          statement1;
          statement2;
          . . .
          statementn;
      }
  }
```

- *public* indicates it is visible from any point of the program.
- *static* is to call the method without creating an instance (object) of the start class.
- *void* indicates, it does not return anything.
- *String args[ ]* is to take the data from command prompt.

Compilation and Execution



```
class TestMain
{ public static void
  main(String[] argv)
  { double x = 2.5;
    x = x * x
  }
}
```

Compiler

```
0101010
1011001
0100101
0100100
1010100
1001010
1001010
```

Java Virtual Machine

Machine dependent executing program

Source Code

Java Byte Code

The Scope and Lifetime of Variables

Scope

In what parts of the program the variable is visible

Lifetime

How much time the value of the variable exists

- Java allows variables to be declared within any block.
- A block begins with an opening curly brace and ends by a closing curly brace.
- A block defines a *scope***.** Thus, each time you start a new block, you are creating a new scope.

Contd..

- In Java, there are major scopes which are defined by a class and a method.

Scopes defined by a method

- The scope defined by a method begins with its opening curly brace and ends with closing curly brace.
- Scopes can be nested.
- Objects declared in the outer scope will be visible to code within the inner scope. The reverse is not true.
- Objects declared within the inner scope will not be visible outside it.

Ex:-

```
class Scopelife
{
  public static void main(String args[])
  {
     int i=20;
     {
        int j=30;
        System.out.print("i="+i+",j="+j);
        {
           int k=10;
           {
             System.out.print("i="+i+",j="+j+",k="+k);
           }
        }
     }
  }
}
```

Operators

Arithmetic Operators

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

- The operands of the arithmetic operators must be of a numeric type.
- You cannot use them on **boolean** types, but you can use them on **char** types.
- No fractional component attached to the result, when the division operator is applied to an integer type.
- The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Ex:- int x = 42;
double y = 42.25;

x mod 10 = 2
y mod 10 = 2.25

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other.
- 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

Note :-

```
int done=1;
// ...
if(!done) ... // Valid in C/C++
if(done) ... // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0)) ... // This is Java-style.
if(done != 0) ...
```

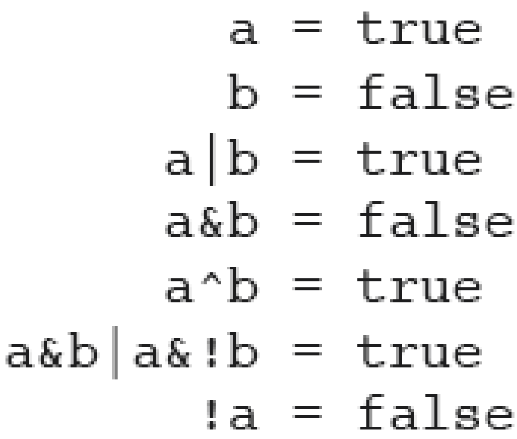
- In C/C++, true is any nonzero value and false is zero.
- In Java, **true** and **false** are nonnumeric values which do not relate to zero or nonzero.

Boolean Logical Operators

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

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

If result is determined by left operand itself, no need to evaluate right hand operand



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

Ex:- && (Short-circuit AND)

```
        if (denom != 0 && num / denom > 10)
```

The ? Operator

General form:

expression1 **?** *expression2* **:** *expression3 ;*

- Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.

The Assignment Operator

- The *assignment operator* is the single equal sign, **=**.

var = *expression*;

- Here, the type of *var* must be compatible with the type of *expression.*
- It allows you to create a chain of assignments.

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

The Bitwise Operators

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

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right (or signed shift right)
>>>	Shift right zero fill (unsigned shift right)
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment

Cotnd..

^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

a = 0011 1100
b = 0000 1101

----

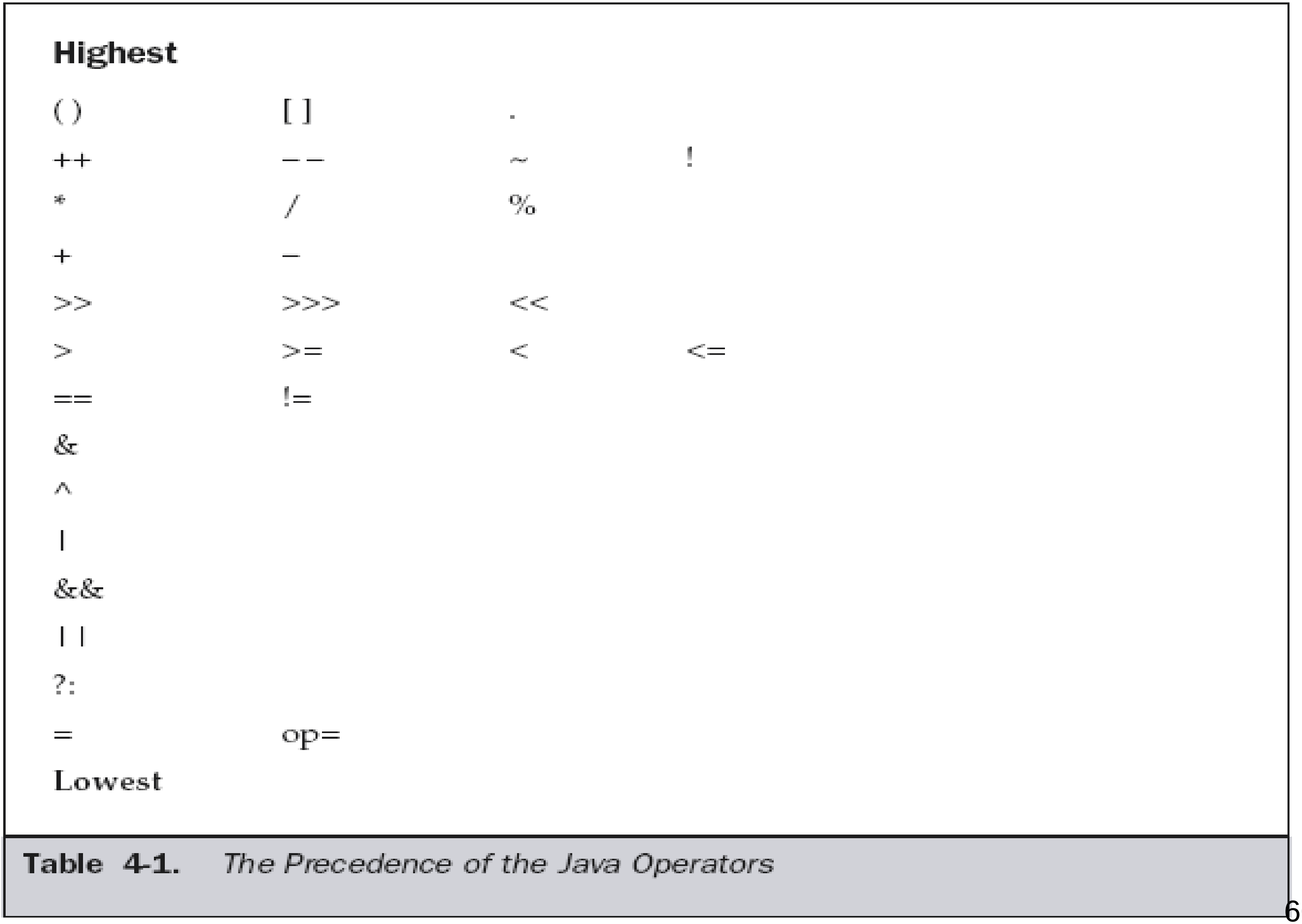
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

Example:

Note: If use >> operator, If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler. (It always fills 0 irrespective of the sign of the number, if use >>> operator)

```
int x = -4;
x = x >> 1;        // x = -2
int y = 4;
y = y >> 1;    //y = 2
int z;
z = 20>>2   //   z = 5
z = 20>>>2  // z = 5
z = -20>>2  // z = -5
z = -20>>>2 // z = 1073741819
```

Operator Precedence



Highest

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

Lowest

Table 4-1. *The Precedence of the Java Operators*

Control Statements

Java’s control statements can be put into the following categories:

- ➢ selection
- ➢ iteration
- ➢jump

Java’s Selection Statements

Java supports two selection statements:

- **if (condition)**
  - Same as in c/c++.
  - The *condition* is any expression that returns a **boolean** value.

- **Switch(expression)**
  - Same as in c/c++.
  - The *expression* must be of type **byte**, **short**, **int**, or **char**;
  - Each **case** value must be a constant, not a variable.
  - Each of the *values* specified in the **case** statements must be of a type compatible with the expression.

Iteration Statements

Java’s iteration statements are

- **for**
- **while**
- **do-while**.

- Syntax is similar to c/c++.
- The *condition* is any expression that returns a **boolean** value.

Jump Statements

Java supports three jump statements:

1. break

- First, it terminates a statement sequence in a **switch** statement.

- Using break to Exit a Loop

  break;

- Using break as a Form of Goto

  label :

  - - - -

  - - - -

  break *label*;

Ex:-

```
// Using break as a advanced form of goto.
class Break
{
    public static void main(String args[ ])
    {
    boolean t = true;
    first: {
    second: {
          third: {
                    System.out.println("Before the break.");
                    if(t) break second; // break out of second block
                    System.out.println("This won't execute");
                 }
                    System.out.println("This won't execute");
             }
                    System.out.println("This is after second block.");
          }
    }
}
```

- Running this program generates the following output:

```
Before the break.
This is after second block.
```

Note : labeled break works only for nested blocks.

2. continue

- **Similar to c/c++**

3. return

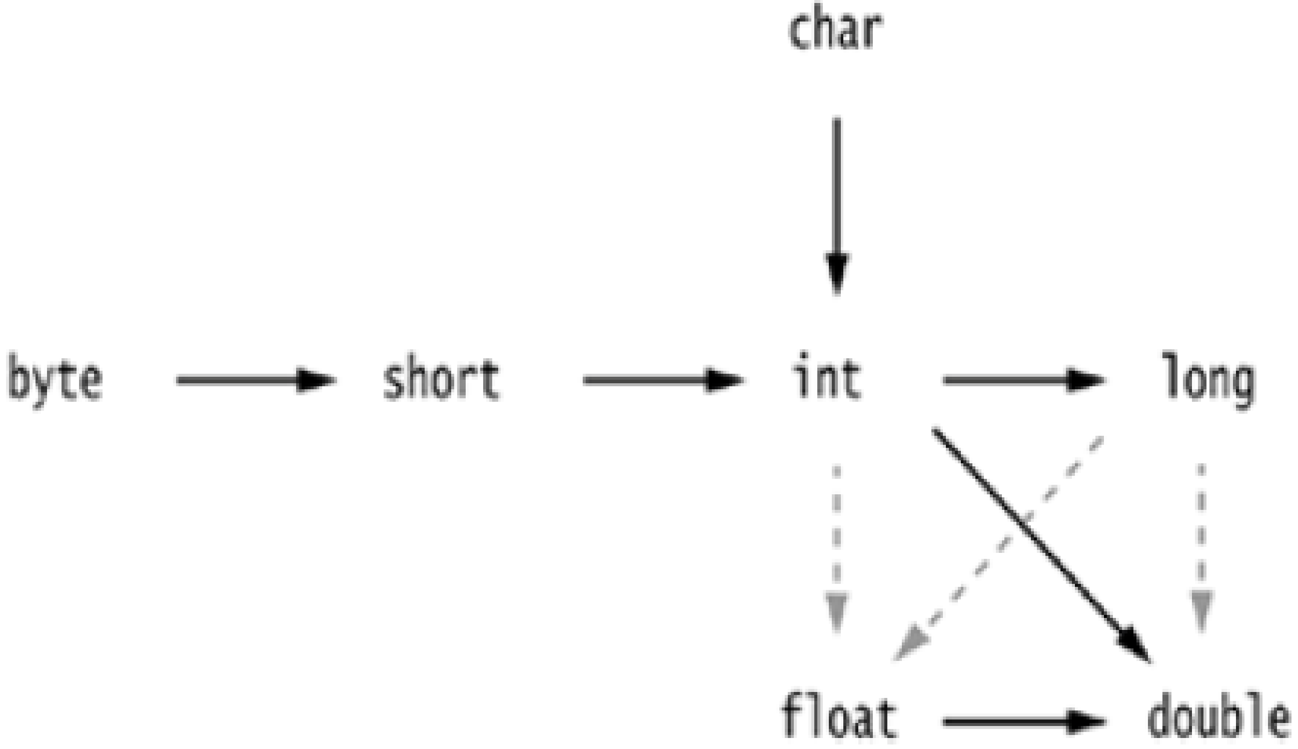
- The **return** statement is used to explicitly return from a method.
- It transfers control back to the caller of the method.

Type Conversion and Casting

Java’s Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.
- When these two conditions are met, a *widening conversion* takes place.
- For widening conversions, the numeric types, including integer, floating-point, and double types, are compatible with each other.
- Also, <u>**char** and **boolean** are not compatible</u> with each other.

- The following fig shows the automatic conversions of primitive data types



- Six solid arrows denote conversions without *information loss.*
- Dotted arrows denote conversions that may *lose information.*
- For example, a large integer such as 123456789 has more digits than the *float* type can represent.
- When the integer is converted to a float, the resulting value has the *correct* magnitude, but it loses some *precision*.

```
Ex :   int n=123456789;
       float  f=n;      //  1.23456792E8;
```

- When an expression contains only integer type of values, first, all **byte** and **short** values are promoted to **int.**

- Then, if one operand is a **long**, the whole expression is promoted to **long.**

- If one operand is a **float,** the entire expression is promoted to **float**.

- If one operand is a **double**, the entire expression is promoted to **double**.

Casting (*Explicit Conversion*)

General form:

```
( targrt-type ) value;
```

- This kind of conversion is sometimes called a *narrowing conversion.*

Ex:

```
float f=3.4f;
int x=( int ) f;     //  x=3
```

Examples:

i)

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

ii)

```
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 = 0.1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}
```

Arrays

Note :- *Arrays creation in Java is different from c/c++.*

One- Dimensional Arrays :-

Method-I	Method II

int month_days[ ];
month_days =new type[size];
	int month_days[ ]=new type[size];

Ex:-	Ex:-

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

- We can obtain the length of an array by using *length* property.

  ex:- int x[ ]=new int[5];

  int len=x.length; length is 5

- Default values of an array are zero’s.
- Direct initialization

  int x[ ]={ 1,2,3 };

Multidimensional Arrays :-

- *It is an array of arrays.*

Method-I	Method II

int twoD[ ][ ];
twoD =new type[row_size][col_size];
	int twoD[ ][ ]=new type[row_size][col_size];

Ex:-	Ex:-

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

Manually allocate differing size second dimensions

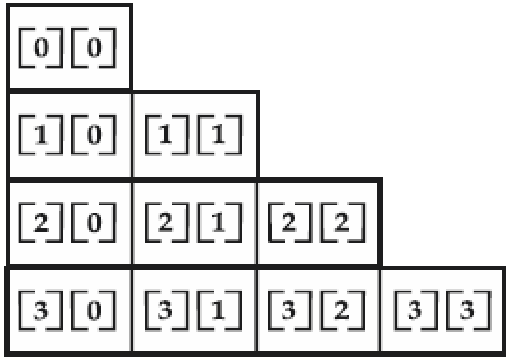
- When you allocate memory for a multidimensional array, you need to specify the memory for the first (leftmost) dimension only.
- We can allocate the remaining dimensions separately.

Ex:-

```
    int twoD[ ][ ] = new int[4][ ];
    twoD[0] = new int[5];
    twoD[1] = new int[5];
    twoD[2] = new int[5];
    twoD[3] = new int[5];
```

- The length of each row is under your control

```
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];
```



- Direct initialization

```
int twoD[ ][ ] ={     { 1,2,3 },
                      {4,5,6,7 }
                 };
```

Alternative Array Declaration Syntax

type[ ] *var-name;*

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

Same

```
int  twoD[ ][ ] = new char[3][4];
int[ ][ ]  twoD = new char[3][4];
```

Same

Class

- ➢ It is a template or blueprint that describes the data (state) and behavior associated with objects of that **class**

Object

- ➢ It is an instance of a class

Example:

Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

The General Form of a 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
    }
}
```

- The data, or variables, defined within a class are called ***instance variables***.
- Functions defined within a class are called ***methods***.
- The methods and variables defined within a class are called ***members*** of the class.

Note:-

The class declaration and the implementation of the methods should be within the class.

it means class declaration and the implementation of the methods are stored in the same place and not defined separately.

Declaring Objects

Ex:-

```
class Box
{
    double width;
    double height;
    double depth;
}

Box mybox = new Box();
```

Example

```
class Box
{
    double width;
    double height;
    double depth;
}
class BoxDemo
{
    public static void main(String args[ ])
    {
        Box mybox = new Box();
        double vol;
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

Constructors

- A ***constructor*** initializes an object immediately upon creation.
- It has the ***same name*** as the class name and is ***syntactically*** similar to a ***method***.
- Constructors do not have any return type, not even ***void.***

```
class Box
{
        double width;
        double height;
        double depth;

        Box()
        {
                width = 10;
                height = 10;
                depth = 10;
        }
}
```

Parameterized Constructors

- In the previous example, all boxes have the ***same dimensions***.
- Parameterized constructor are used to construct ***Box objects*** of various dimensions.

```
class Box
{
        double width;
        double height;
        double depth;

        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
}
```

The this Keyword

- ***this*** keyword is used to refer to the *current* object.

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Instance Variable Hiding

- In java, it is illegal to declare *two local variables* with the *same name* inside the block or nested blocks.
- Interestingly, you can have *local variables*, including formal *parameters* to methods, which overlap with the names of the class' *instance variables*.
- When a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- That is why **width**, **height**, and **depth** are not used as the names of the parameters to the **Box( )** constructor inside the **Box** class.

Contd..

- Therefore **,this** can be used to resolve any name collisions that might occur between instance variables and local variables**.**

```
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 C++ , dynamically allocated objects must be manually released by use of a **delete** operator.
- Java handles deallocation automatically. This technique is called *garbage collection.*

  Technique

  - 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**.**
- Garbage collection only occurs at regular intervals during the execution of your program.

Introducing Access Control

- We can control what parts of a program can access the members of a class by using *access specifiers*.

- Java offers four access specifiers.

  - public
  - protected
  - default (no specifier)
  - private

Private:

- A private member is accessible only to the class in which it is defined. Inheritance does not apply on the private members. They are Just like secrets.
- Use `private` keyword to create private members.

Protected:

- Allows the class itself, subclasses, and all classes in the same package to access the members.
- To declare a protected member, use the keyword `protected`.

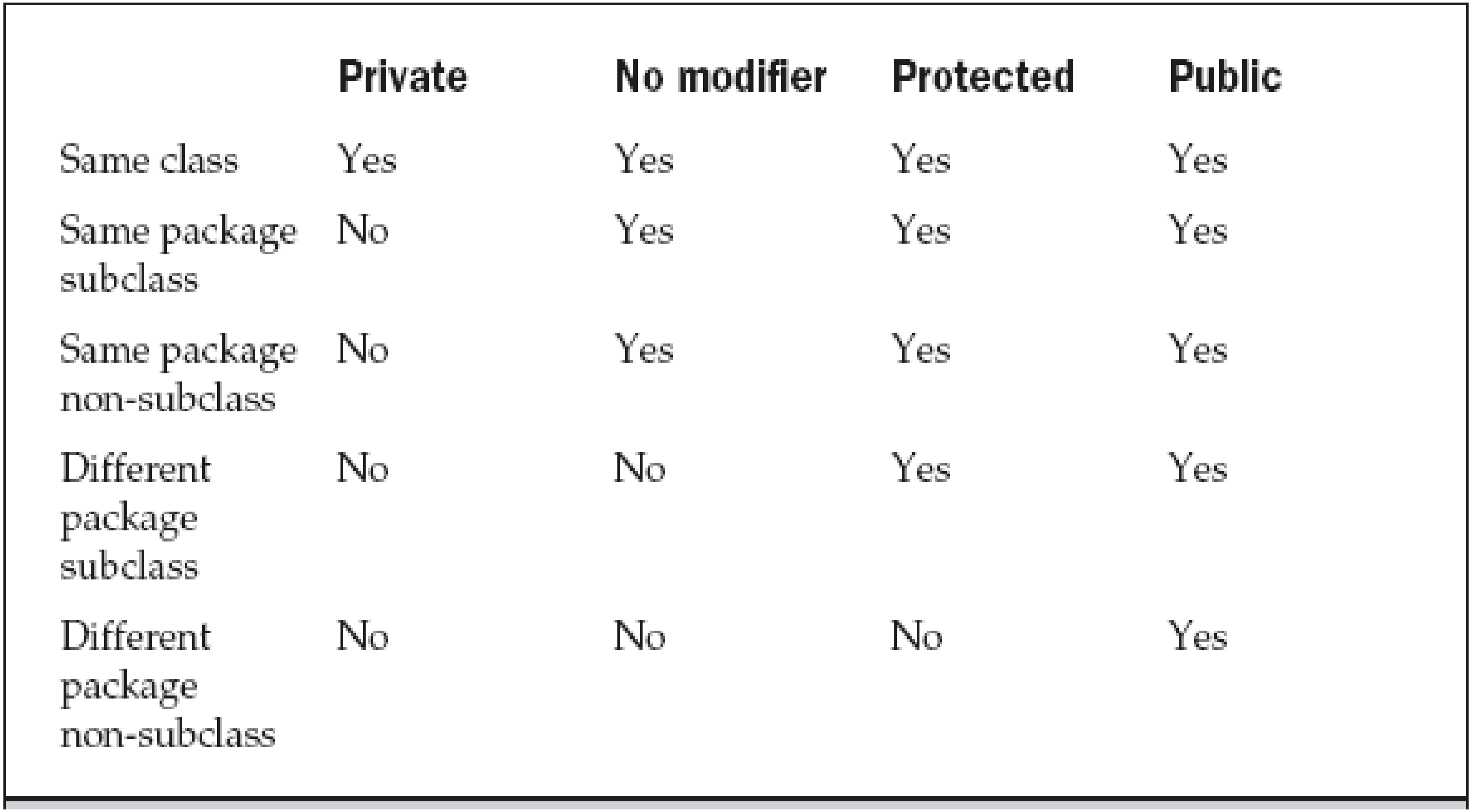
Public:

- Any class, in any package ,has access to a class's public members.
- To declare a public member, use the keyword public.

Default :

- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

To summarize



	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

Class member access

Example:-

```
class Test
{
        int a;
        public int b;
        private int c;
        void setc(int i)
        {
                c = i;
        }
        int getc()
        {
                return c;
        }
}
class AccessTest
{
        public static void main(String args[])
        {
                Test ob = new Test();
                ob.a = 10;
                ob.b = 20;
                ob.c = 100; // Error!
                ob.setc(100); // OK
                System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
        }
}
```

Overloading Methods

- Defining two or more methods within the same class that share the same name is called *method overloading.*

- Java uses the ***number*** or ***type*** or ***order*** of parameters to determine which version of the overloaded method to call

- We can not overload a method based on ***return*** type

Ex :-

```
void println()
void println(boolean x)
void println(char x)
void println(double x)
void println(float x)
```

Example:-

```
class OverloadDemo
{
    void test()
    {
            System.out.println("No parameters");
    }
    void test(int a)
    {
            System.out.println("a: " + a);
    }
    void test(int a, double b)
    {
    System.out.println("a and b: " + a + " " + b);
    }
    void test(double a, int b)
    {
    System.out.println("a and b: " + a + " " + b);
    }
    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;
            ob.test();
            ob.test(10);
            ob.test(10, 2.67);
            ob.test(1.34, 2);
            result = ob.test(123.25);
            System.out.println("Result of
    ob.test(123.25): " + result);
    }
}
```

Overloading Constructors

- Constructors can be overloaded.

```
class Box
{
        double width;
        double height;
        double depth;

        Box ( )
         {
                width = 10;
                height =15;
                depth = 20;
        }
        Box (double w, double h, double d )
         {
                width = w;
                height = h;
                depth = d;
        }
}
```

Parameter Passing

- There are two common ways that a computer language can pass a parameter to a method.

Call-by-value

- The *call-by-value* copies the *value* of a actual parameter into the formal parameter of the method.
- In this method, changes made to the formal parameter of the method have no effect on the actual parameter.

Call-by-reference

- In *call-by-reference,* a reference of an actual parameter (not the value of the argument) is passed to the formal parameter.
- In this method, changes made to the formal parameter will affect the actual parameter used to call the method.
  - Simple types are passed by value.
  - Objects are passed by reference (including *arrays* and *Strings*)

Recursion

- A method that calls itself is called *recursion.*

Ex:-

```
class Factorial
{
    int fact(int n)
    {
           if(n==1)
              return 1;
           else
             return n*fact(n-1);
    }
}
```

String Handling:-

String Class

- The ***String*** class supports several constructors.

- String is a class not an array of characters in java.

- ***String*** class is defined in ***java.lang*** package

Constructors

- String s = new String(); //Creates an empty string.

- String s= new String(char *chars*[ ])

  Ex:-

  ```
  char chars[ ] = { 'a', 'b', 'c' };
  String s = new String(chars);
  ```

- String(char *chars*[ ], int *startIndex*, int *numChars*)

  Ex:-

  ```
  char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  String s = new String(chars, 2, 3);          s is  "cde"
  ```

➢ We can also create string by using string literal.

- String s2 = "abc";

some of the important methods of the String class

- **`int length()`**
- **`String toUpperCase()`**
- **`String toLowerCase()`**
- **`String substring(int beginIndex)`**
  - `Returns a string containing the characters from beginIndex up to end of the string.`
- **`String substring(int beginIndex, int endIndex)`**
  - `Returns a string containing the characters from beginIndex up to but not including the index endIndex.`
- **`char charAt(int index)`**
- **`boolean equals(Object anotherObject)`**
- **`int compareTo(Object anotherObject)`**
- **`boolean endsWith(String suffix)`**
- **`boolean startsWith(String prefix)`**

```
class  StringExample     {
    public static void main(String[] args)        {
     char chars[ ] =  {'a','b','c','d'};
     String  s1  =  new String(chars);
     System.out.println(s1);
     String s2  =  new String();
     System.out.println(s2);
     String s3  =  new String("abcde");
     System.out.println(s3);
     String s4  =  new String("abe");
     System.out.println(s4);
     System.out.println(s3.length());
     System.out.println(s3.toUpperCase());
     System.out.println(s3.toLowerCase());
     System.out.println(s3.substring(2));
     System.out.println(s3.substring(1,3));
     System.out.println(s3.charAt(2));
     System.out.println(s3.equals(s4));
     System.out.println(s3.compareTo(s4));
     System.out.println(s3.endsWith("e"));
     System.out.println(s3.startsWith("a"));   }   }
```

Understanding static

- Normally a class member must be accessed only through an object of its class.

- However, it is possible to create a member that can be accessed by using a class name.

- **static** keyword will be used to create such a member

- You 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.

- static instance variable is initialized to ***zero*** whenever class is defined.

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

- Instance variables declared as **static will** act as global variables.

- All instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions: (in case of class without object)

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

➢Common use is to keep a count of *how many instances* of a class have been created.

Syntax :

```
class classname
{
     access_specifier  static type variable;
                            :
     access_specifier  static type fun_name()
    {
                  .    .    .
    }
                            :
 }
```

Note :

➢Use of *static* keyword in a *method* is an error.

➢In other words, class members can be static, but local variables can't.

Static block / Initializer

- We have used constructor to initialize instance variables.

- Constructors can not be used to initialize static instance variables as these are common to all the objects of the same class.

- Java provides a feature called a *static initializer / block* that's designed specifically to initialize static instance variables.

- A **static** *initializer / block* is executed whenever the class is loaded.

Syntax :

```
static
{
        statement1;
        statement2;
        …….
        …….
}
```

Ex :

```
class StaticInit
{
          static int x;

         static
         {
                   x = 32;
         }

}
```

- This example is pretty simple.

- In fact, we can achieve the same effect by assigning 32 to the variable when it is declared.

- However,
  - Suppose, if you had to perform a *complicated calculation* to determine the value of x, or
  - Suppose its value comes from a *database.*
  - In that case, a static initializer/block can be very useful.

```
// Demonstration of static variables, methods, and blocks.
//Example1
public class UseStatic {
        static int a = 3;
        static int b;
        static void meth(int x) {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }

   static {
        System.out.println("Static block initialized.");
        b = a * 4;
        }
   public static void main(String args[]) {
   meth(42);
   }
}
```

```
//Example2

class Cls   {
    static int j;
    static void Print()  {
         System.out.println("This is from Cls Print method");
    }
}

public class UseStatic  {
    public static void main(String args[]) {
         System.out.println("j = " + Cls.j);
         Cls.Print();
      }
}
```

StringTokenizer

- In java, there is a way to break up a line of text into what are called tokens

- A token is a smaller piece of a string

- StringTokenizer is used to break up a line of text into tokens

- The **StringTokenizer constructors:**

StringTokenizer(String *str);*

StringTokenizer(String *str, String delimiters);*

StringTokenizer(String *str, String delimiters, boolean delimAsToken);*

Ex:

String str ="title=Java: The Complete Reference;author=Schildt;publisher=McGraw-Hill;copyright=2002";

1.StringTokenizer(*str);*

O/P :

title=Java:
The
Complete
Reference;author=Schildt;publisher=McGraw-Hill;copyright=2002

2. StringTokenizer(*str,";");*

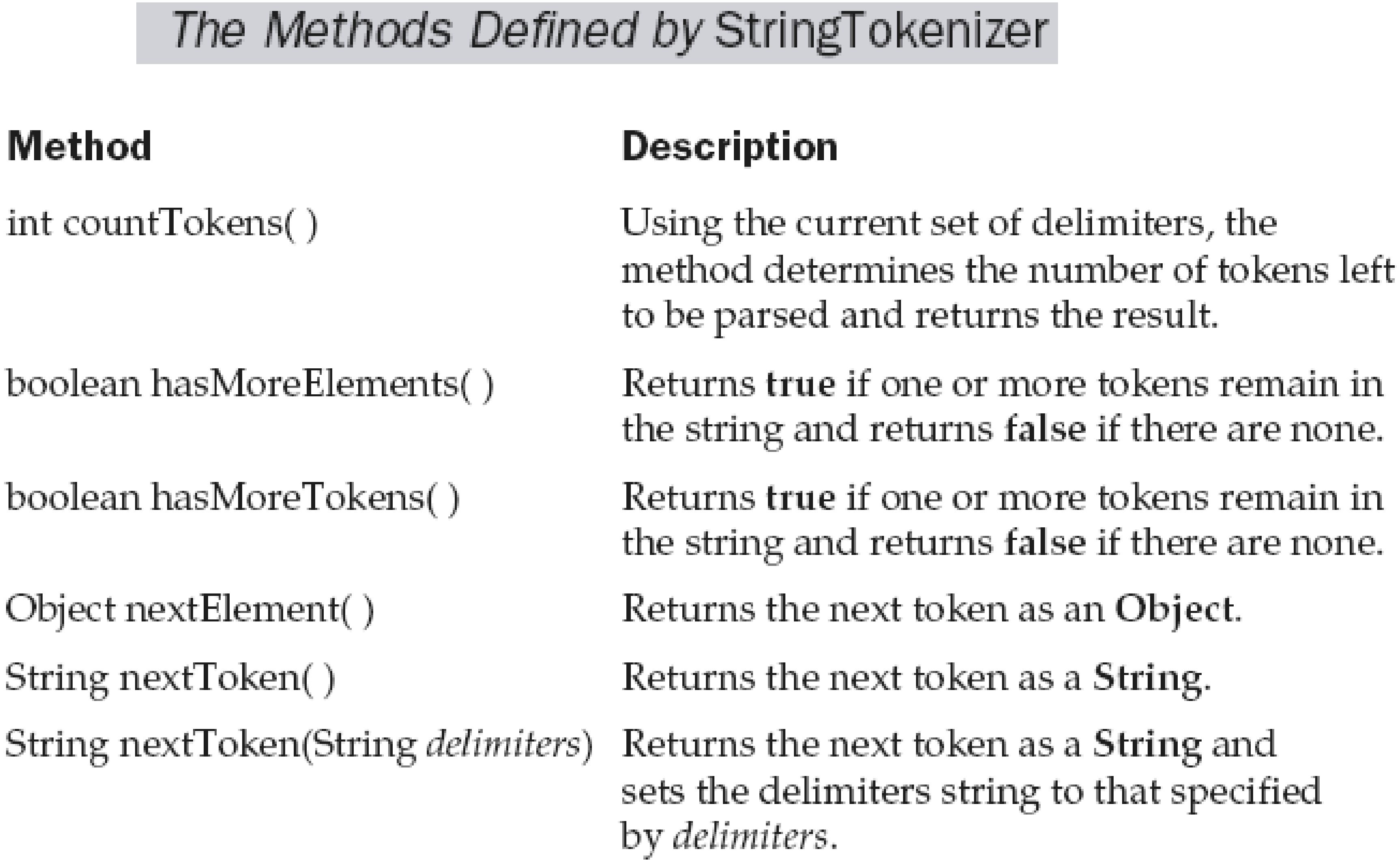
O/P:

title=Java: The Complete Reference
author=Schildt
publisher=McGraw-Hill
copyright=2002

3. StringTokenizer(*str,";" ,true);*

O/P:

```
title=Java: The Complete Reference
;
author=Schildt
;
publisher=McGraw-Hill
;
copyright=2002
```



The Methods Defined by StringTokenizer

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

Ex:

```
import java.util.StringTokenizer;
class STDemo
{
  public static void main(String args[])
  {
    String in = "Sreenidhi Institute of Science and Technology";
    StringTokenizer st = new StringTokenizer(in," ");
    while(st.hasMoreTokens())
    {
      String t = st.nextToken();
      System.out.println(t);
    }
  }
}
```

Output:

```
Sreenidhi
Institute
of
Science
and
Technology
```

Types of variables:

1. Local variables
2. Instance variables
3. Class/static variables

Default values of Local variables

-They do not have default values
-You should initialize local variables before you are using them, otherwise compiler gives error

Default values of instance variables and static variables

-for integers 0 (byte, short, int and long)
-for floats 0.0 (float and double)
-for boolean false
-for objects null
-for character empty or blank (not null)