



## Lesson Objectives

After completing this lesson, participants will be able to:

- Understand Basic Java Language constructs like:
  - Keywords
  - Primitive Data Types
  - Operators
  - Variables
  - Literals
- Write Java programs using control structures
- Best Practices



## Lesson Outline:

- 3.1: Keywords
- 3.2: Primitive Data Types
- 3.3: Operators and Assignments
- 3.4: Variables and Literals
- 3.5: Flow Control: Java's Control Statements
- 3.6: Best Practices

## 7.1 : Keywords

## Keywords in Java

abstract	continue	for	new	switch
assert <sup>***</sup>	default	goto <sup>*</sup>	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum <sup>****</sup>	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp <sup>**</sup>	volatile
const <sup>*</sup>	float	native	super	while

Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. E.g. class, boolean, abstract, do, try etc. Incorrect usage results in compilation errors.

In addition, three identifiers are reserved as predefined literals in the language: null, true and false.

The table above shows the keywords available in Java 5.

## 7.2: Primitive Data types

## Java Data types

Type	Size/Format	Description
byte	8-bit	Byte-length integer
short	16-bit	Short Integer
int	32-bit	Integer
long	64-bit	Long Integer
float	32-bit IEEE 754	Single precision floating point
double	64-bit IEE 754	Double precision floating point
char	16-bit	A single character
boolean	1-bit	True or False

There are two data types available in Java:

Primitive Data Types.

Reference/Object Data Types :- is discussed later.

There are eight primitive data types supported by Java (see slide above). Primitive data types are predefined by the language and named by a key word.

The default character set used by Java language is **Unicode character** set and hence a **character data type** will consume **two bytes** of memory instead of a byte (a standard for ASCII character set). Unicode is a character coding system designed to support text written in diverse human languages.

This allows you to use characters in your Java programs from various alphabets such as Japanese, Greek, Russian, Hebrew, and so on. This feature **supports** a readymade support for **internalization** of java.

The default values for the various data types are as follows:

Integer : 0  
Character : '\u0000'  
Decimal : 0.0  
Boolean : false  
Object Reference: null

*[Note: C or C++ data types pointer, struct, and union are not supported. Java does not have a typedef statement (as in C and C++).]*

7.3 : Operators and Assignments

## Operators in Java

Operators can be divided into following groups:

- Arithmetic
- Bitwise
- Relational
- Logical
- *instanceof* Operator

Java provides a rich set of operators to manipulate variables. These are classified into several groups as shown above.

## 7.3 : Operators and Assignments

## Arithmetic Operators

Operator	Result
+	Addition
-	Subtraction (or unary) operator
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
--	Decrement

Arithmetic operators are summarized in the table above:

Integer division yields an integer quotient for example, the expression  $7 / 4$  evaluates to 1, and the expression  $17 / 5$  evaluates to 7. Any fractional part in integer division is simply discarded (i.e., truncated) no rounding occurs.

Java provides the remainder operator, %, which yields the remainder after division. The expression  $x \% y$  yields the remainder after  $x$  is divided by  $y$ . Thus,  $7 \% 4$  yields 3, and  $17 \% 5$  yields 2. This operator is most commonly used with integer operands, but can also be used with other arithmetic types.

Parentheses are used to group terms in Java expressions in the same manner as in algebraic expressions. For example, to multiply  $a$  times the quantity  $b + c$ , we write  **$a*(b+c)$** .

If an expression contains nested parentheses, such as  **$((a+b)*c)$**  the expression in the innermost set of parentheses ( $a + b$  in this case) is evaluated first.

**Order of Precedence:**

Multiplication, division and remainder operations are applied first. If an expression contains several such operations, the operators are applied from left to right.

**Multiplication, division and remainder operators have the same level of precedence.**

Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from **left to right**. **Addition and subtraction operators have the same level of precedence.**

## 7.3 : Operators and Assignments

## Bitwise Operators

Apply upon *int*, *long*, *short*, *char* and *byte* data types:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
=	Bitwise OR assignment

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types. The operators discussed in this section are less commonly used.

The unary bitwise complement operator "~" inverts a bit pattern; it can be applied to any of the integral types, making every "0" a "1" and every "1" a "0". For example, a byte contains 8 bits; applying this operator to a value whose bit pattern is "00000000" would change its pattern to "11111111".

The signed left shift operator "<<" shifts a bit pattern to the left, and the signed right shift operator ">>" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator ">>>" shifts a zero into the leftmost position, while the leftmost position after ">>" depends on sign extension.

Bitwise & operator performs a bitwise AND operation.

Bitwise ^ operator performs a bitwise exclusive OR operation.

Bitwise | operator performs a bitwise inclusive OR operation.

## 7.3 : Operators and Assignments

## Relational Operators

Determine the relationship that one operand has to another.

- Ordering and equality.

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

A condition is an expression that can be either true or false. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If the condition in an if statement is true, the body of the if statement executes. If the condition is false, the body does not execute.

Conditions in if statements can be formed by using the equality operators (== and !=) and relational operators (>, <, >= and <=). Both equality operators have the same level of precedence, which is lower than that of the relational operators. The equality operators associate from left to right. The relational operators all have the same level of precedence and also associate from left to right.



## 7.3 : Operators and Assignments

## Logical Operators

Operator	Result
&&	Logical AND
	Logical OR
^	Logical XOR
!	Logical NOT
==	Equal to
?:	Ternary if-then-else

Java provides logical operators to enable programmers to form more complex conditions by combining simple conditions. The logical operators are && (conditional AND), || (conditional OR), & (boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (logical NOT).

## 7.4: Variables and Literals

## Variables

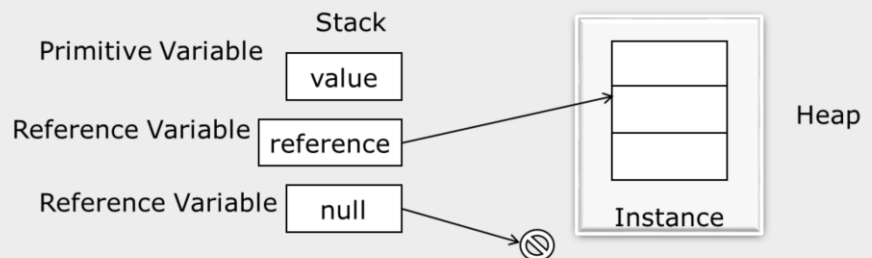
Variables are data placeholders.

Java is a strongly typed language, therefore every variable must have a declared type.

The variables can be of two types:

- reference types: A variable of reference type provides a reference to an object.
- primitive types: A variable of primitive type holds a primitive.

In addition to the data type, a Java variable also has a name or an identifier.



## 7.4: Variables and Literals



## Types of Variables

Variable is basic storage in a Java program

Three types of variables:

- Instance variables
  - Instantiated for every object of the class
- Static variables
  - Class Variables
  - Not instantiated for every object of the class
- Local variables
  - Declared in methods and blocks

**Instance variables:** These are members of a class and are instantiated for every object of the class. The values of these variables at any instant constitute the *state* of the object.

**Static variables:** These are also members of a class, but these are not instantiated for any object of the class and therefore belong only to the class. We shall be covering the static modifier in later section.

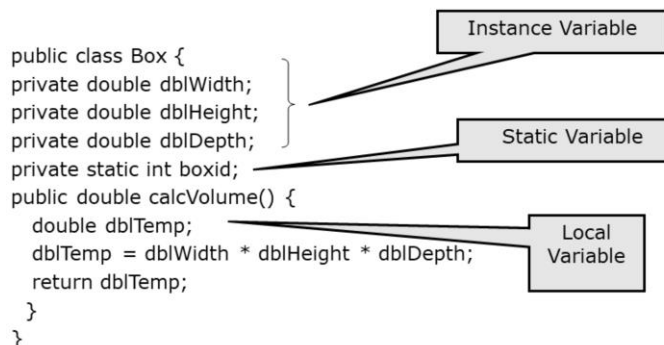
**Local variables:** These are declared in methods and in blocks. They are instantiated for every invocation of the method or block. In Java, local variables must be declared before they are used.

Life-cycle of the variable is controlled by the scope in which those are defined.

Refer the example on the subsequent slide.

## 7.4: Variables and Literals

## Types of Variables



```
public class Box {  
    private double dblWidth;  
    private double dblHeight;  
    private double dblDepth;  
    private static int boxid;  
    public double calcVolume() {  
        double dblTemp;  
        dblTemp = dblWidth * dblHeight * dblDepth;  
        return dblTemp;  
    }  
}
```

Add the notes here.

## 7.4: Variables and Literals



## Literals

Literals represents value to be assigned for variable.

Java has three types of literals:

- Primitive type literals
- String literals
- null literal

Primitive literals are further divided into four subtypes:

- Integer
- Floating point
- Character
- Boolean

For better readability of large sized values, Java 7 allows to include '\_' in integer literals.

Literal Type	Example
Integer	int x = 10
Octal	int x = 0567
Hexadecimal	int x = 0x9E (to represent number 9E)
Long	long x = 9978547210L;
Binary	byte twelve = 0B1100; (to represent decimal 12)
Using Underscores	int million = 1_000_000; int twelve = 0B_1100; long multiplier = 12_34_56_78_90_00L;
Float	float x = 0.4f; float y = 1.23F, float z = 0.5e10;
Double	double x = 0.0D; double pi=3.14; double z=9e-9d;
Boolean	boolean member=true; boolean applied=false;
Character	char gender = 'm';
String	String str = "Hello World";
Null	Employee emp = null;

7.5: Flow Control: Java's Control Statements



## Control Statements

Use control flow statements to:

- Conditionally execute statements
- Repeatedly execute a block of statements
- Change the normal, sequential flow of control

Categorized into two types:

- Selection Statements
- Iteration Statements

Java being a programming language, offers a number of programming constructs for decision making and looping.

## 7.5: Flow Control: Java's Control Statements

## Selection Statements

Allows programs to choose between alternate actions on execution.

"if" used for conditional branch:

```
if (condition) statement1;  
else statement2;
```

"switch" used as an alternative to multiple "if's":

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

**Expression can be  
of String type!**

**If Statement:**

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

Each statement may be a single statement or a compound statement enclosed in curly braces. The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed.

Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

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

## 7.5: Flow Control: Java's Control Statements

## switch case : an example

```
class SampleSwitch {  
    public static void main(String args[]) {  
        for(int i=0; i<=4; 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.");  
            }  
    }  
}
```

**Output:**  
i is zero.  
i is one.  
i is two.  
i is three.  
i is greater than  
3.

**Switch – Case:**

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

The switch-expression must evaluate to any of the following type:

- byte
- short
- char
- int
- enum
- **String.**

As such, it often provides a better alternative than a large series of *if-else-if* statements.



## 7.5: Flow Control: Java's Control Statements

## Iteration Statements

Allow a block of statements to execute repeatedly

- While Loop: Enters the loop if the condition is true

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

- Do - While Loop: Loop executes at least once even if the condition is false

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

**while statement:**

The body of the loop is executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. Example:

```
class Samplewhile {
    public static void main(String args[]) {
        int n = 5;
        while(n > 0) {
            System.out.print(n+"\t");
            n--;
        }
    }
}
```

**Output:** 5 4 3 2 1

The while loop evaluates its conditional expression at the top of the loop. Hence, if the condition is false to begin with, the body of the loop will not execute even once.

**do – while loop:**

This construct executes the body of a **while** loop at least once, even if the conditional expression is false to begin with. The termination expression is tested at the **end** of the loop rather than at the beginning.

## 7.5: Flow Control: Java's Control Statements

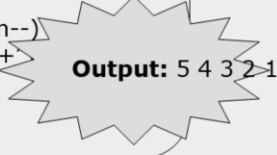
## Iteration Statements

## ▪ For Loop:

```
for( initialization ; condition ; iteration)
{ //body of the loop }
```

## ▪ Example

```
// Demonstrate the for loop.
class SampleFor {
    public static void main(String args[]) {
        int number;
        for(number =5; number >0; n--)
            System.out.print(number + " ")
        }
    }
```



**Output: 5 4 3 2 1**

**for loop:**

When the **for** loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. The initialization expression is only executed once. Next, *condition* is evaluated. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed, else the loop terminates. Next, the *incremental* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

7.5: Control Structures

## Demo

Data types in Java

Switch Statement using String as expression



Add the notes here.



7.6: Best Practices

### Best practices: Iteration Statements

Always use an int data type as the loop index variable whenever possible

Use for-each liberally

Switch case statement

Terminating conditions should be against 0

Loop invariant code motion

E.g If you call length() in a tight loop, there can be a performance hit.

**Always use an int data type as the loop index variable whenever possible.**

It is efficient when compared to using byte or short data types. This is because when we use byte or short data type as the loop index variable they involve implicit type cast to int data type.

**Use for-each liberally.**

The for-each loop is used with both collections and arrays. It is intended to simplify the most common form of iteration, where the iterator or index is used solely for iteration, and not for any other kind of operation, such as removing or editing an item in the collection or array.

When there is a choice, the for-each loop should be preferred over the for loop, since it increases legibility.

**Switch Case statement.**

One peculiar fact about switch statements is that code consisting of case with consecutive constants like 0,1,2 are faster than with case with constants like 2, 6, 7, 14, etc. This is because in the former switching between options requires less offset and it takes only 16 bytes. But in the later the offset is higher and it might take 32 or more bytes.

## Summary

In this lesson you have learnt:

- Keywords
- Primitive Data Types
- Operators and Assignments
- Variables and Literals
- Flow Control: Java's Control Statements
- Best Practices



### Review Question

Question 1: Java considers variable number and NuMbEr to be identical.

- True/False

Question 2: The *do...while* statement tests the loop-continuation condition \_\_\_\_\_ it executes executing the loop's body; hence, the body executes at least once.

- **Option1:** before
- **Option2:** after

