Java Programming

Introduction to Java Programming

Module 1

Agenda

1. Evolution of Java
2. Java Architecture
3. Langugage Basics
4. Flow Control

Objectives

At the end of this module, you will be able to :

- Learn about Evolution of Java and forces that shaped it
- Understand Java Architecture along with JVM Concepts
- Write the first Java Program with understanding of Language Basics and Keywords
- Learn about how control the program execution flow using branching and looping constructs

Evolution of Java

Key Founders

- Java was the brainchild of:
  - James Gosling
  - Patrick Naughton
  - Chris Warth
  - Ed Frank &
  - Frank Sheridan
- The origin of Java can be traced back to the fall of 1992, and was initially called **Oak**
- Oak was renamed as **Java** in **1995**

Design Goal

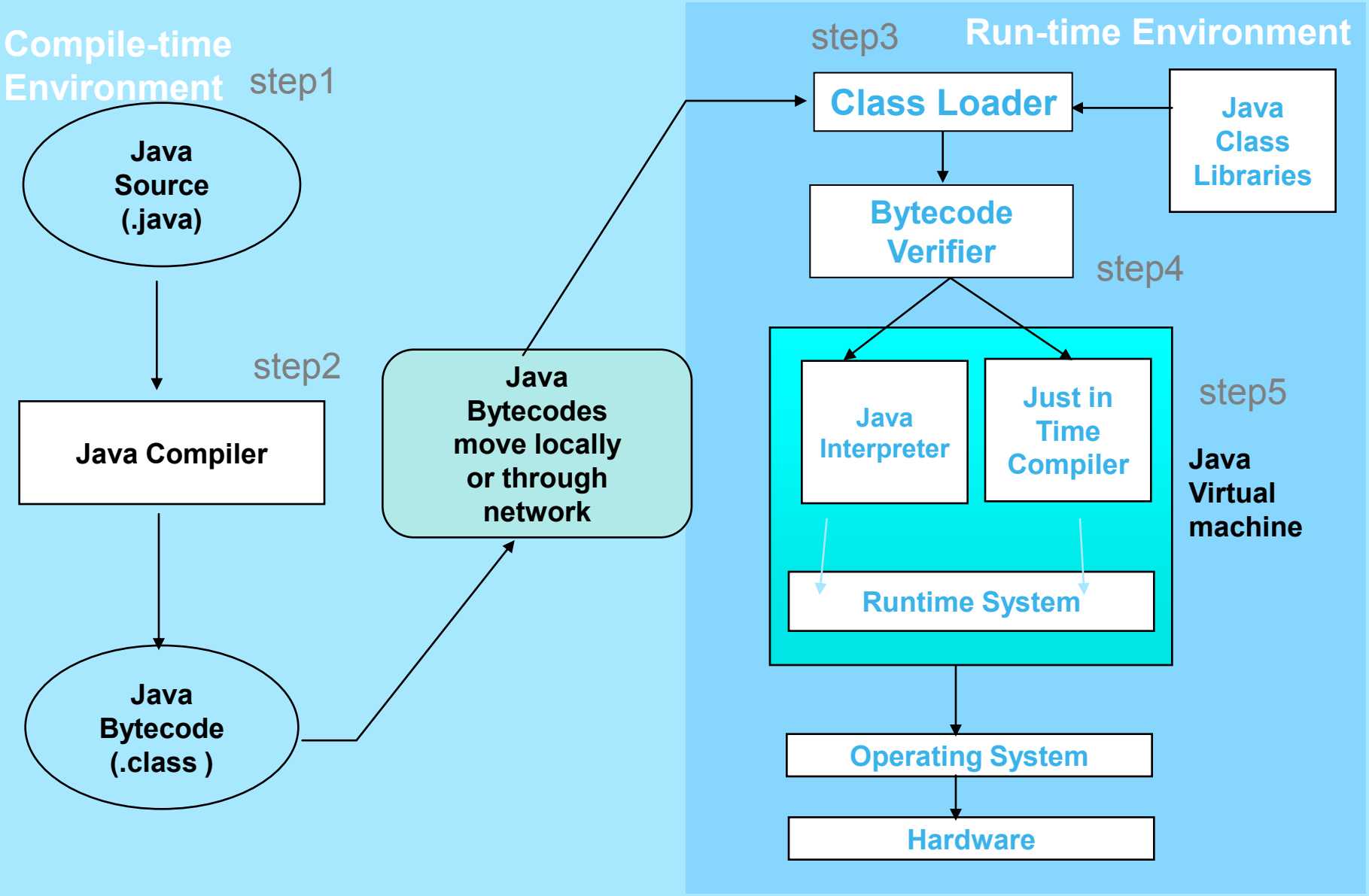
- Java was originally meant to be a platform-neutral language for embedded software in devices

- The goal was to move away from platform and OS-specific compilers that would compile source for a particular target platform to a language that would be portable, and platform-independent

- The language could be used to produce platform-neutral code

Java Architecture

Java Architecture



Java Architecture (Contd.).

Step1:

Create a java source code with .java extension

Step2:

Compile the source code using java compiler, which will create bytecode file with .class extension

Step3:

Class loader reads both the user defined and library classes into the memory for execution

Java Architecture (Contd.).

Step4:

Bytecode verifier validates all the bytecodes are valid and do not violate Java's security restrictions

Step5:

JVM reads bytecodes and translates into machine code for execution. While execution of the program the code will interact to the operating system and hardware

The 5 phases of Java Programs

Java programs can typically be developed in five stages:

1. Edit

   Use an editor to type Java program (**Welcome.java**)
2. Compile

   - Use a compiler to translate Java program into an intermediate language called bytecodes, understood by Java interpreter (**javac Welcome.java)**
   - Use a compiler to create **.class** file, containing bytecodes (**Welcome.class**)
3. Loading

   Use a class loader to read bytecodes from **.class** file into memory

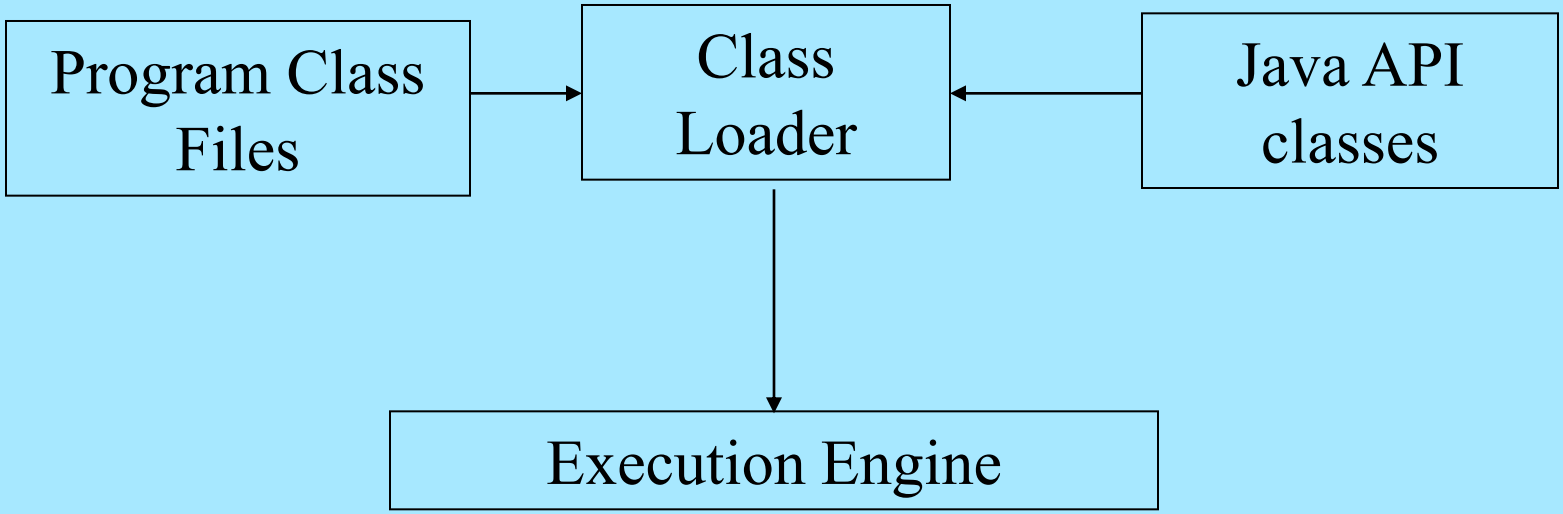
The 5 phases of Java Programs (Contd.).

4. Verify

   Use a Bytecode verifier to make sure bytecodes are valid and do not violate security restrictions

5. Execute
   - Java Virtual Machine (JVM) uses a combination of interpretation and just-in-time compilation to translate bytecodes into machine language
   - Applications are run on user's machine, i.e. executed by interpreter with java command (java Welcome)

The Java Architecture – The JVM



The Java Architecture – The JVM (Contd.).

- Most modern languages are designed to be compiled
- Compilation is a one-time exercise and executes faster
- Execution of compiled code over the Internet an impossibility
- Executable code always generated to a CPU-OS combination
- Interpreting a Java program into byte code facilitates its execution in a wide variety of environments

The Java Architecture – The JVM (Contd.).

- Only the Java Virtual Machine (JVM) needs to be implemented for each platform
- Once the Java runtime package exists for a given system, any Java program can run on it
- The JVM will differ from platform to platform, and is, platform-specific
- All versions of JVM interpret the same Java byte code

The Java Architecture – The JVM (Contd.).

- Interpreted code runs much slower compared to executable code
- The use of bytecode enables the Java runtime system to execute programs much faster
- Java facilitates on-the-fly compilation of bytecode into native code

The Java Architecture – The Adaptive optimizer

- Another type of execution engine is an adaptive optimizer
- The virtual machine starts by interpreting bytecodes
- It also keeps a tab on the code that is running and identifies only the heavily used areas
- The JVM compiles these heavily used areas of code into native code
- The rest of the code, which is not heavily used continues to be interpreted and executed

The Java Architecture - The Class Loader

- The class loader is that part of the VM that is important from:
  - A security standpoint
  - Network mobility

- The class loader loads a compiled Java source file (**.class** files represented as bytecode) into the Java Virtual Machine (JVM)

- The bootstrap class loader is responsible for loading the classes, programmer defined classes as well as Java's API classes

The Java Architecture - The Java .class file

- The Java class file is designed for
  - platform independence
  - network mobility

- The class file is compiled to a target JVM, but independent of underlying host platforms

- **The Java class file is a binary file** that has the capability to run on any platform

Quiz

1. Write the correct order of the Java program execution

A. Class Loader
B. Interpretation
C. Compilation
D. Byte Code Verification
E. Java Source Code
F. Execution

2. A Sample.Java file contains, class A, B and C. How many .class files will be created after compiling Sample.java?

Langugage Basics

A Simple Java Program

A First Java Program:

```
public class Welcome {
  public static void main(String args[]) {
   System.out.println("Welcome to Java
  Programming");
  }
}
```

- Create source file: Welcome.java
- Compile : javac Welcome.java
- Execute : java Welcome

The Java API

- An application programming interface(API), in the framework of java, is a collection of prewritten packages, classes, and interfaces with their respective methods, fields and constructors
- The Java API, included with the JDK, describes the function of each of its components
- In Java programming, many of these components are pre-created and commonly used

The Java Buzzwords

- Simple
- Object-Oriented
  - Supports encapsulation, inheritance, abstraction, and polymorphism
- Distributed
  - Libraries for network programming
  - Remote Method Invocation
- Architecture neutral
  - Java Bytecodes are interpreted by the JVM

The Java Buzzwords (Contd.).

- Secure
  - Difficult to break Java security mechanisms
  - Java Bytecode verification
  - Signed Applets
- Portable
  - Primitive data type sizes and their arithmetic behavior specified by the language
  - Libraries define portable interfaces
- Multithreaded
  - Threads are easy to create and use

Language Basics

- Keywords
- Data Types
- Variables
- Operators
- Conditional Statements
- Loops

Java Keywords

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

Primitive Data Types

Data Type	Size (in bits)	Minimum Range	Maximum Range	Default Value (for fields)
byte	8	-128	+127	0
short	16	-32768	+32767	0
int	32	-2147483648	+2147483647	0
long	64	-9223372036854775808	+9223372036854775807	0L
float	32	1.40E-45	3.40282346638528860e+38	0.0f
double	64	4.94065645841246544e-324d	1.79769313486231570e+308d	0.0d
char	16		0 to 65,535	'\u0000'
boolean	1	NA	NA	false

Types of Variables

The Java programming language defines the following kinds of Variables:

- Local Variables
  - Tied to a method
  - Scope of a local variable is within the method
- Instance Variables (Non-static)
  - Tied to an object
  - Scope of an instance variable is the whole class
- Static Variables
  - Tied to a class
  - Shared by all instances of a class

Quiz

1. Match the followig table:

DATA TYPES	SIZE(bytes)
char	4
byte	2
int	1
double	8

2. What will be the output for the below code ?

```
public class Sample{
public static void main(String a[]){
int i_val;
System.out.println("i_val value: "+i_val);
}

}
```

Operators

- Java provides a set of operators to manipulate operations.
- Types of operators in java are,
  - Arithmetic Operators
  - Unary Operator
  - Relational Operators
  - Logical Operators
  - Simple Assignment Operator
  - Bitwise Operators

Arithmetic Operators

The following table lists the arithmetic operators

Operator	Description	Example
+	Addition	A + B
-	Subtraction	A - B
*	Multiplication	A * B
/	Division	A/B
%	Modulus	A%B

Arithmetic Operators - Example

```
/* Example to understand  Arithmetic operator */

class Sample{
public static void main(String[] args){
int a = 10;
int b = 3;
System.out.println("a + b = " + (a + b) );
System.out.println("a - b = " + (a - b) );
System.out.println("a * b = " + (a * b) );
System.out.println("a / b = " + (a / b) );
System.out.println("a % b = " + (a % b) );
}
}
```

Output:

a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1

Unary Operators

The following table lists the unary operators

Operator	Description	Example
+	Unary plus operator	+A
-	Unary minus operator	-A
++	Increment operator	++A or A++
--	Decrement operator	--A or A--

Unary Operator - Example

```
/* Example for Unary Operators*/
class Sample{
public static void main(String args[]) {
    int a = 10;
    int b = 20;

    System.out.println("a++    = " +   (++a) );
    System.out.println("b--    = " +   (--b) );
    }
}
```

Output:

a++ = 11
b-- = 19

Relational Operators

The following table lists the relational operators

Operator	Description	Example
==	Two values are checked, and if equal, then the condition becomes true	(A == B)
!=	Two values are checked to determine whether they are equal or not, and if not equal, then the condition becomes true	(A != B)
>	Two values are checked and if the value on the left is greater than the value on the right, then the condition becomes true.	(A > B)
<	Two values are checked and if the value on the left is less than the value on the right, then the condition becomes true	(A < B)
>=	Two values are checked and if the value on the left is greater than equal to the value on the right, then the condition becomes true	(A >= B)
<=	Two values are checked and if the value on the left is less than equal to the value on the right, then the condition becomes true	(A <= B)

Relational Operators - Example

```
/* Example to understand  Relational operator */

class Sample{
public static void main(String[] args){
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
    System.out.println("b <= a = " + (b <= a) );
}
}
```

Output:

a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false

Logical Operators

The following table lists the logical operators

Operator	Description	Example
&&	This is known as Logical AND & it combines two variables or expressions and if and only if both the operands are true, then it will return true	(A && B) is false
||	This is known as Logical OR & it combines two variables or expressions and if either one is true or both the operands are true, then it will return true	(A || B) is true
!	Called Logical NOT Operator. It reverses the value of a Boolean expression	!(A && B) is true

Logical Operators - Example

```
/* Example to understand  logical operator */

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

Output:

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

Simple Assignment Operator

= Simple assignment operator

Which assigns right hand side value to left hand side variable

Ex:

```
int a;
a = 10;
```

Quiz

- What will be the output for the below code ?

```
public class Sample{
   public static void main(){
      int i_val = 10, j_val = 20;
      boolean chk;
      chk = i_val < j_val;
      System.out.println("chk value: "+chk);
   }
}
```

Flow Control

Control Statements

- Control statements are statements which alter the normal execution flow of a program

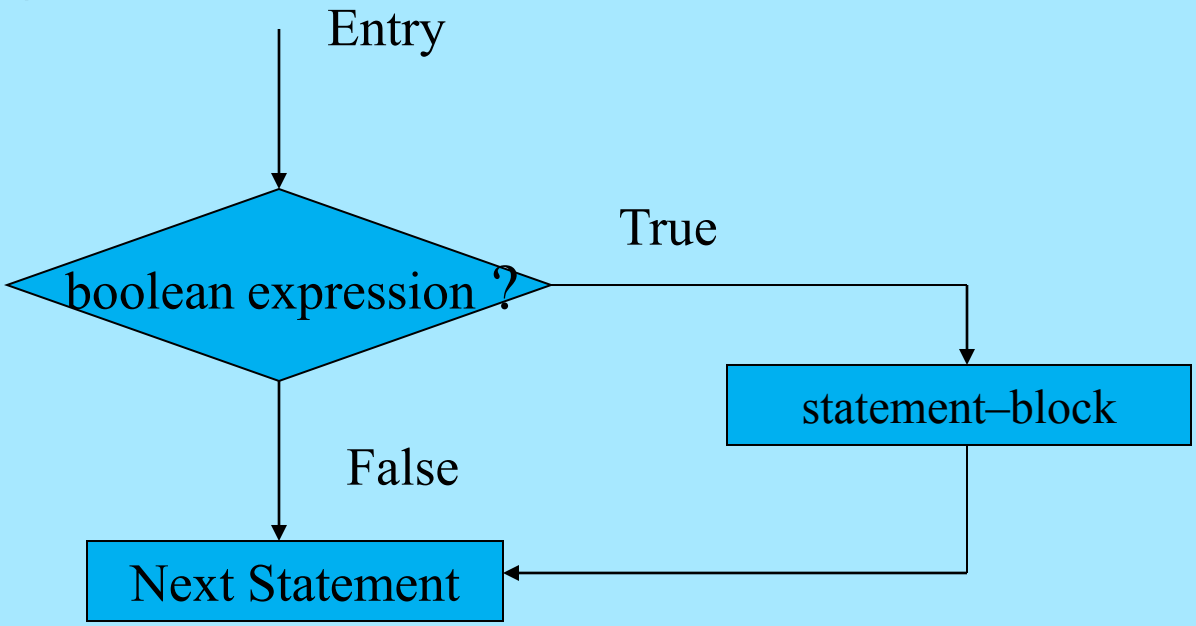
- There are three types of Control Statements in java

Selection statement	Iteration Statement	Jumping Statement
if	while	break
if – else	for	continue
switch	do – while	return

Simple if statement

syntax :

```
if(boolean expression)
{
    statement–block;
 }
 Next statement;
```



If - Example

```
/* This is an example of a if statement */

public class Test {
public static void main(String args[]){
int x = 5;
if( x < 20 ){
System.out.print("This is if statement");
}
}
}
```

Output:

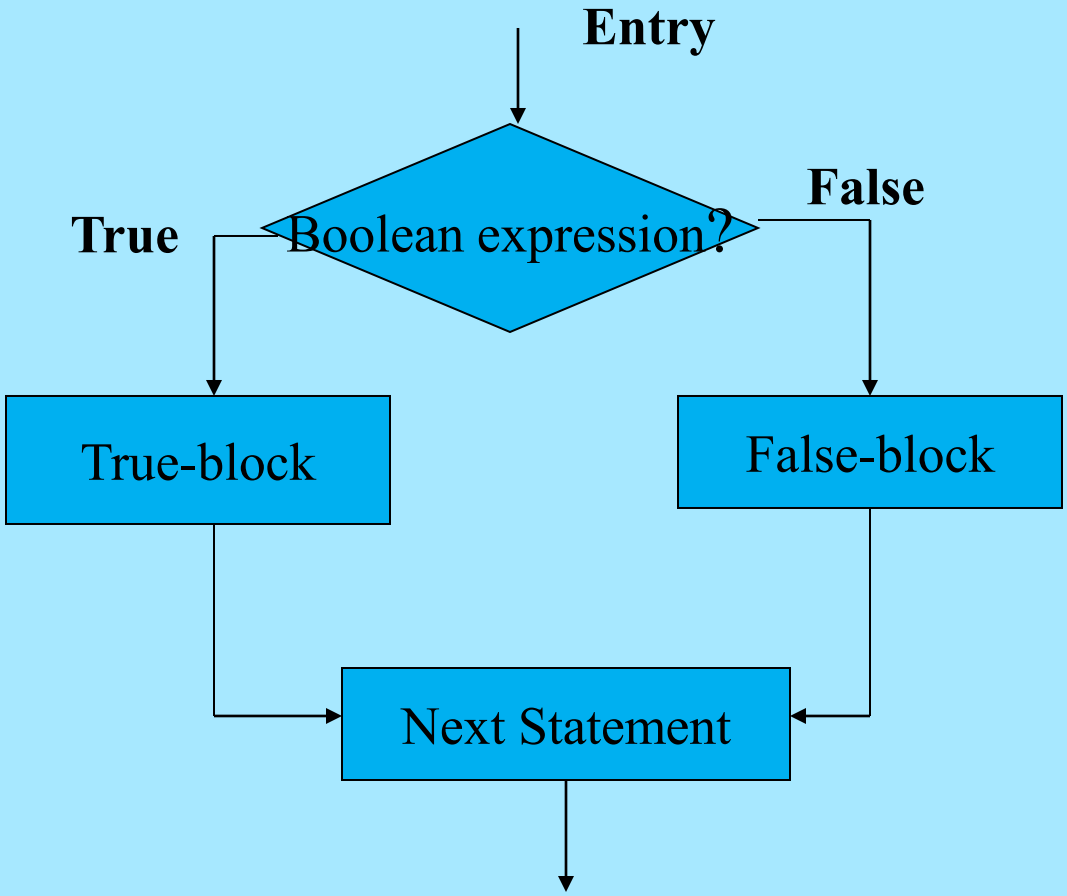
This is if statement

If..else statement

The if….else statement is an extension of simple if statement

Syntax,

```
if(Boolean expression)
 {
     True-block statements;
 }
 else
 {
     False-block statements;
 }
 Next statement;
```



If – else Example

```
/* program to check given age input is eligible to
  vote or not using if- else*/
public class Check {
public static void main(String[ ] args) {
int age;
age = Integer.parseInt(args[0]);
if(age>18) {
  System.out.println("Eligible to vote");
}
else{
  System.out.println("Not eligible to vote");
}
}
}
```

Cascading *if- else*

- **Syntax:**

```
if (condition1)
{
        statement-1
}
….
else if(conditio-n)
{
        statement-n
}
else
{
        default statement
}
next statement
```

else - if Example

```
/* program to print seasons for a month input using else--if
  */

public class ElseIfDemo {
  public static void main(String[] args) {
    int month = Integer.parseInt(args[0]);
    if(month == 12 || month == 1 || month == 2)
        System.out.println("Winter");
    else if(month == 3 || month == 4 || month == 5)
        System.out.println("Spring");
    else if(month == 6 || month == 7 || month == 8)
        System.out.println("Summer");
    else if(month == 9 || month == 10  || month == 11)
        System.out.println("Autumn");
else
        System.out.println("invalid month");
}
}
```

If args[0] is 6 then the Output:
Summer

Switch Case

- The switch-case conditional construct is a more structured way of testing for multiple conditions rather than resorting to a multiple if statement

Syntax:

```
switch (expression)       {
          case value-1 : case-1 block
                              break;
         case value-2 : case-2 block
                              break;
              default  : default block
                              break;
      }
statement-x;
```

Switch Case - Example

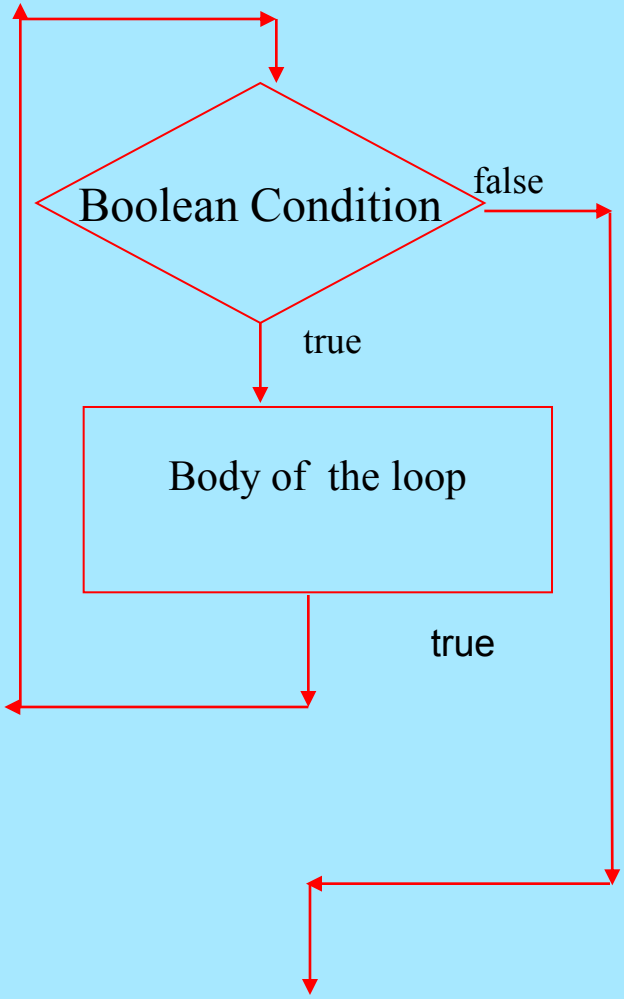
```
/* This is an example of a switch case statement*/
public class SwitchDemo {
public static void main(String[] args) {
int weekday = Integer.parseInt(args[0]);
    switch(weekday) {
    case 1:   System.out.println("Sunday");
           break;
    case 2:   System.out.println("Monday");
           break;
    case 3:   System.out.println("Tuesday");
           break;
    case 4:   System.out.println("Wednesday");
           break;
    case 5:   System.out.println("Thursday");
           break;
    case 6:   System.out.println("Friday");
           break;
    case 7:   System.out.println("Saturday");
           break;
    default:  System.out.println("Invalid day");
    }   }    }
```

If args[0] is 6 then the Output:
Friday

While loop

- **Syntax**

```
while(condition)
{
    Body of the loop
}
```



while loop – Example

```
/* This is an example for a while loop */
public class Sample{
public static void main(String[] args) {
 int i = 0;
  while (i < 5)
{
System.out.println("i: "+i);
i = i + 1;
}
}
}
```

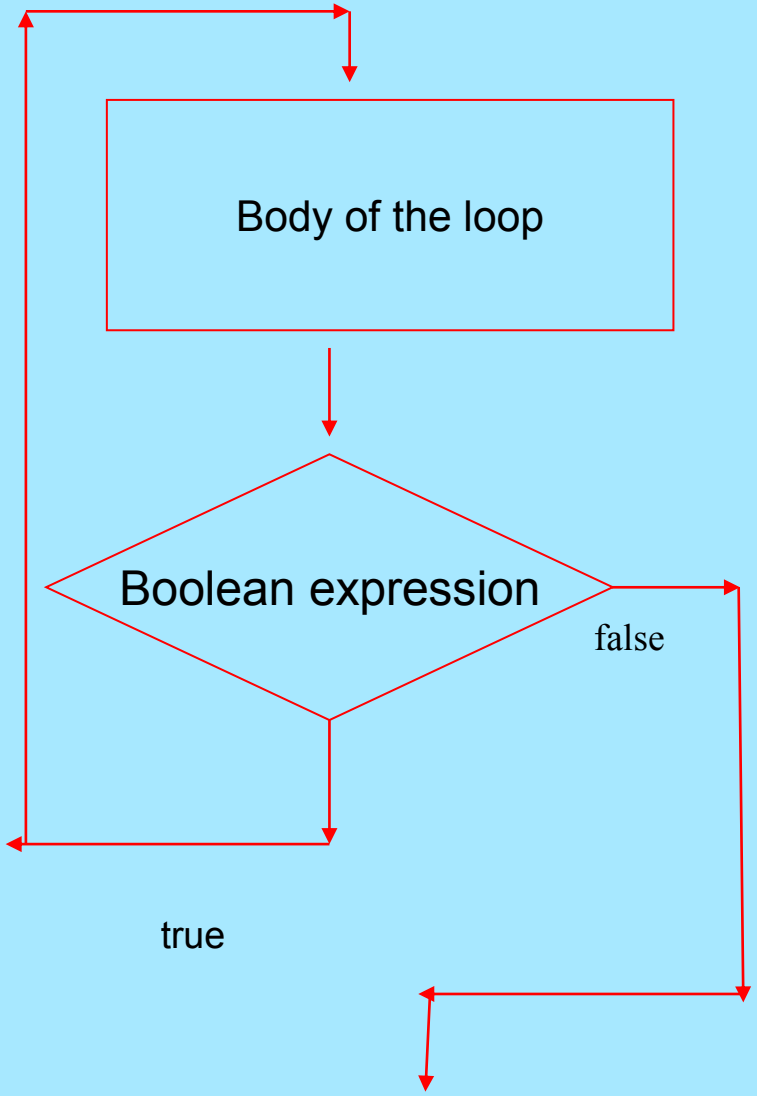
Output:

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

do-while loop

- **Syntax:**

```
do
{
    Body of the loop
} while(boolean expression);
```



do…while loop – Example

```
/* This is an example of a do-while loop */
public class Sample{
public static void main(String[] args) {
 int i =5;
do {
System.out.println("i: "+i);
i = i + 1;
} while (i < 5);
}
}
```

Output:

i: 5

For loop

- **Syntax**

```
for(initialization;condition;increment/decrement)
{
    Body of the loop
}
```

For loop - Example

```
/* This is an example of a for loop */

public class Sample{
public static void main(String[] args) {
 for (int  i=1; i<=5; i++ ) {
  System.out.println("i: "+i);
}
}
}
```

Output:

i: 1
i: 2
i: 3
i: 4
i: 5

Enhanced for loop

- **Syntax:**

```
for(declaration : expression) {
Body of loop
 }
```

Enhanced for loop - Example

```
/* This is an example of a enhanced for loop */

public class Sample{
public static void main(String[] args) {
int [] numbers = {10, 20, 30, 40, 50};
for(int i : numbers ){
System.out.println( "i: "+i );
}
}
}
```

Output:

i:10
i:20
i: 30
i:40
i:50

break statement

- While the execution of program, the break statement will terminate the iteration or switch case block
- When a break statement is encountered in a loop, the loop is exited and the program continues with the statements immediately following the loop
- When the loops are nested, the break will only terminate the corresponding loop body

break - Example

```
/* This is an example of a break statement */


public class Sample{
public static void main(String[] args) {
 for (int  i=1; i<=5; i++ ) {
  if(i==2)
    break;
  System.out.println("i: "+i);
}
}
}
```

Output:

i: 1

continue statement

- The continue statement skips the current iteration of a loop
- In while and do loops, continue causes the control to go directly to the test-condition and then continue the iteration process
- In case of for loop, the increment section of the loop is executed before the test-condition is evaluated

Continue - Example

```
/* This is an example of a continue loop */
public class Sample{
public static void main(String[] args) {
int [] numbers = {1, 2, 3, 4, 5};
for(int i : numbers ){
if( i == 3 ){
continue;
}
System.out.println( "i: "+i );
}
}
}
```

Output:

i: 1
i:2
i:4
i:5

Quiz

- What will be the result, if we try to compile and execute the following code

```
class Sample{
 public static void main(String[]args){
      boolean b = true;
    if(b){
      System.out.println(" if block ");
    }
    else{
      System.out.println(“ else block ");
    }
  }
}
```

Quiz

- What will be the result, if we try to compile and execute the following codes

1.
```
class Sample{
  public static void main(String[] args){
  while(false){
        System.out.println("while loop");
 }}}
```

2.
```
class Sample{
  public static void main(String[] args){
  for(;;){
        System.out.println("For loop");
 }}}
```

Summary

In this session, you were able to :

- Learn about Evolution of Java and forces that shaped it
- Understand Java Architecture along with JVM Concepts
- Write the first Java Program with understanding of Language Basics and Keywords
- Learn about how control the program execution flow using branching and looping constructs

References

1. Oracle (2012). Java Tutorials: Primitive Data Types. Retrieved on May 12, 2012, from, http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
2. Oracle (2012). Java Tutorials: Assignment, Arithmetic, and Unary Operators. Retrieved on May 12, 2012, from, http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op1.html
3. Schildt, H. Java: The Complete Reference. J2SETM. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.
4. Tutorialpoint TP (2012). C-Operator Types. Retrieved on May 12, 2012, from, http://www.tutorialspoint.com/ansi_c/c_operator_types.htm

Thank You