Object Oriented Programming with JAVA

Module 1- An Overview of Java

Dr. Vinoth Kumar M , Associate Professor
Dept. of Information Science & Engineering,
RVITM

Dr. Kirankumar K , Assistant Professor Dept. of Information Science & Engineering,
RVITM

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- **Java Is Secure**
- **Java Is Architecture-Neutral**
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java Platform Editions

- A Java Platform is the set of APIs, class libraries, and other programs used in developing Java programs for specific applications.

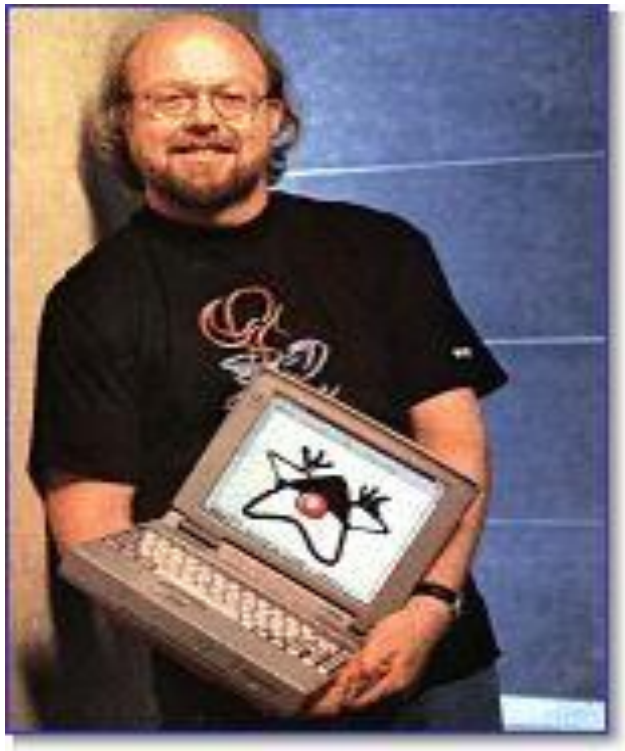
1. Java 2 Platform, Standard Edition (J2SE)
   - Core Java Platform targeting applications running on workstations

2. Java 2 Platform, Enterprise Edition (J2EE)
   - Component-based approach to developing distributed, multi-tier enterprise applications

3. Java 2 Platform, Micro Edition (J2ME)
   - Targeted at small, stand-alone or connectable consumer and embedded devices

James Gosling

- James Gosling is generally credited as the inventor of the Java programming language
- He was the first designer of Java and implemented its original compiler and virtual machine
- He is also known as the Father of Java.

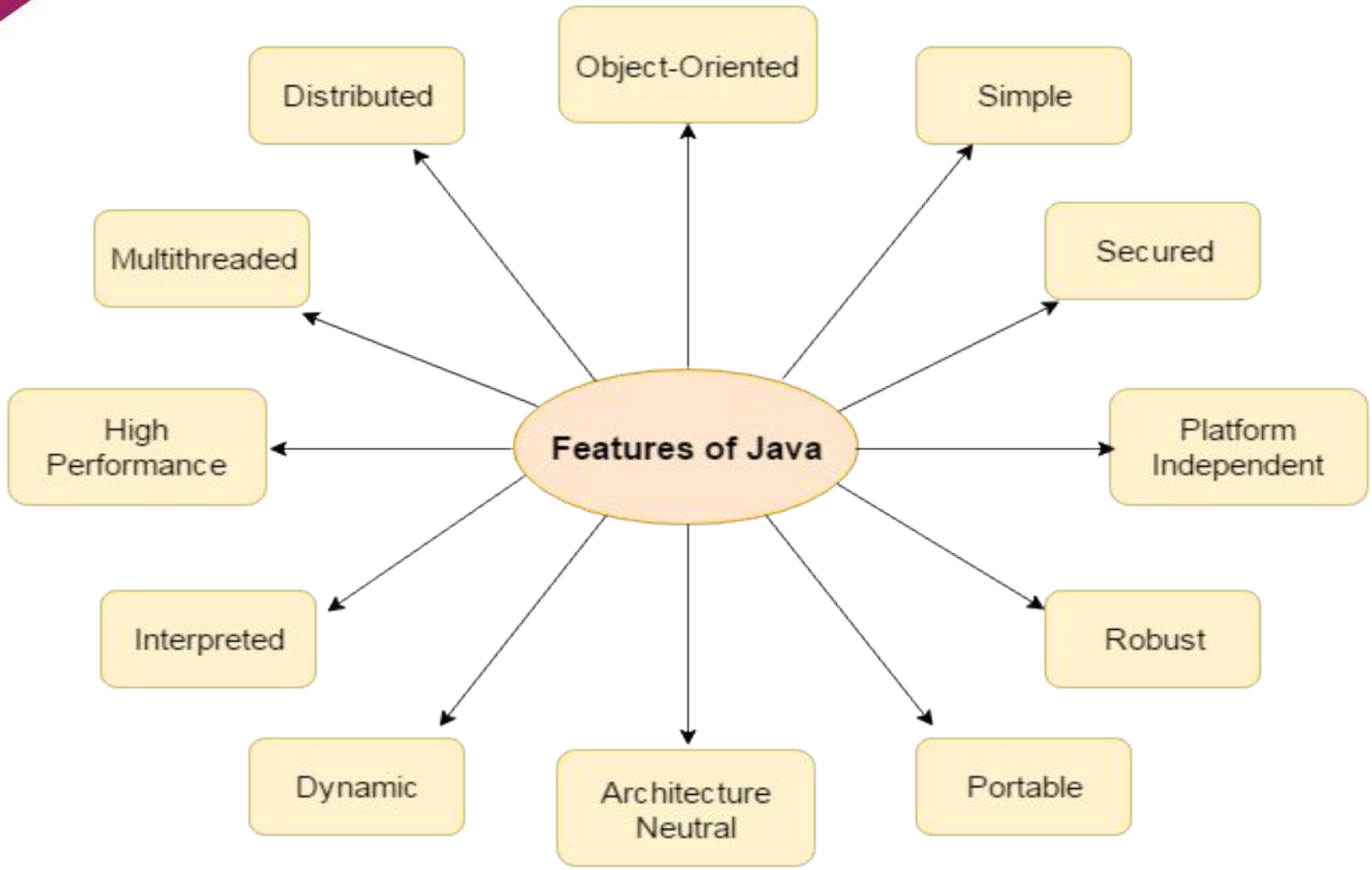


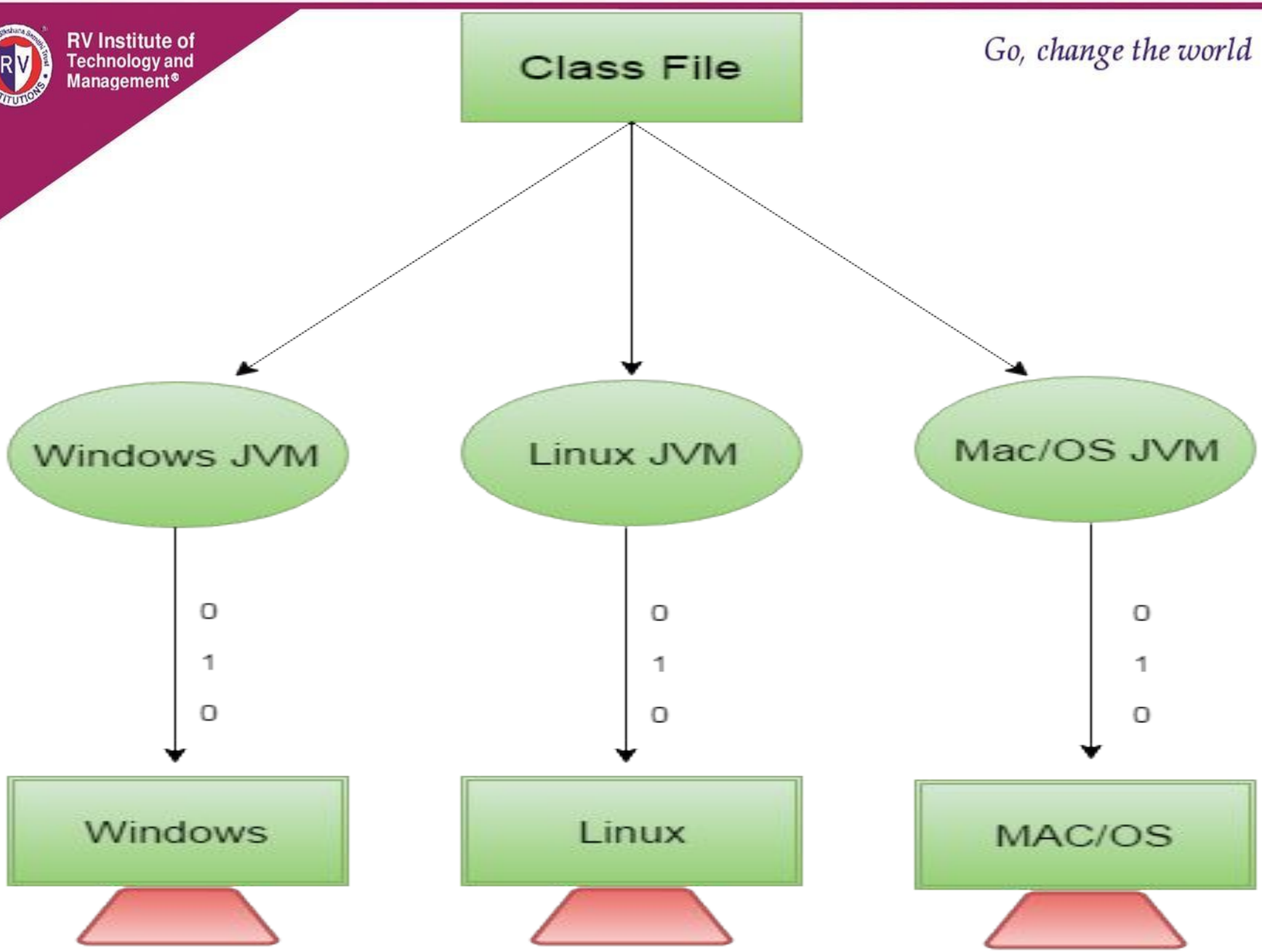
Brief History of Java

- In 1990, Sun Microsystems began an internal project known as the ***Green Project*** to work on a new technology.
- In 1992, the Green Project was spun off and its interest directed toward building highly interactive devices for the cable TV industry. This failed to materialize.
- **In 1994**, the focus of the original team was re-targeted, this time to the use of Internet technology. A small web browser called ***HotJava*** was written.
- Oak was renamed to ***Java*** after learning that Oak had already been trademarked.

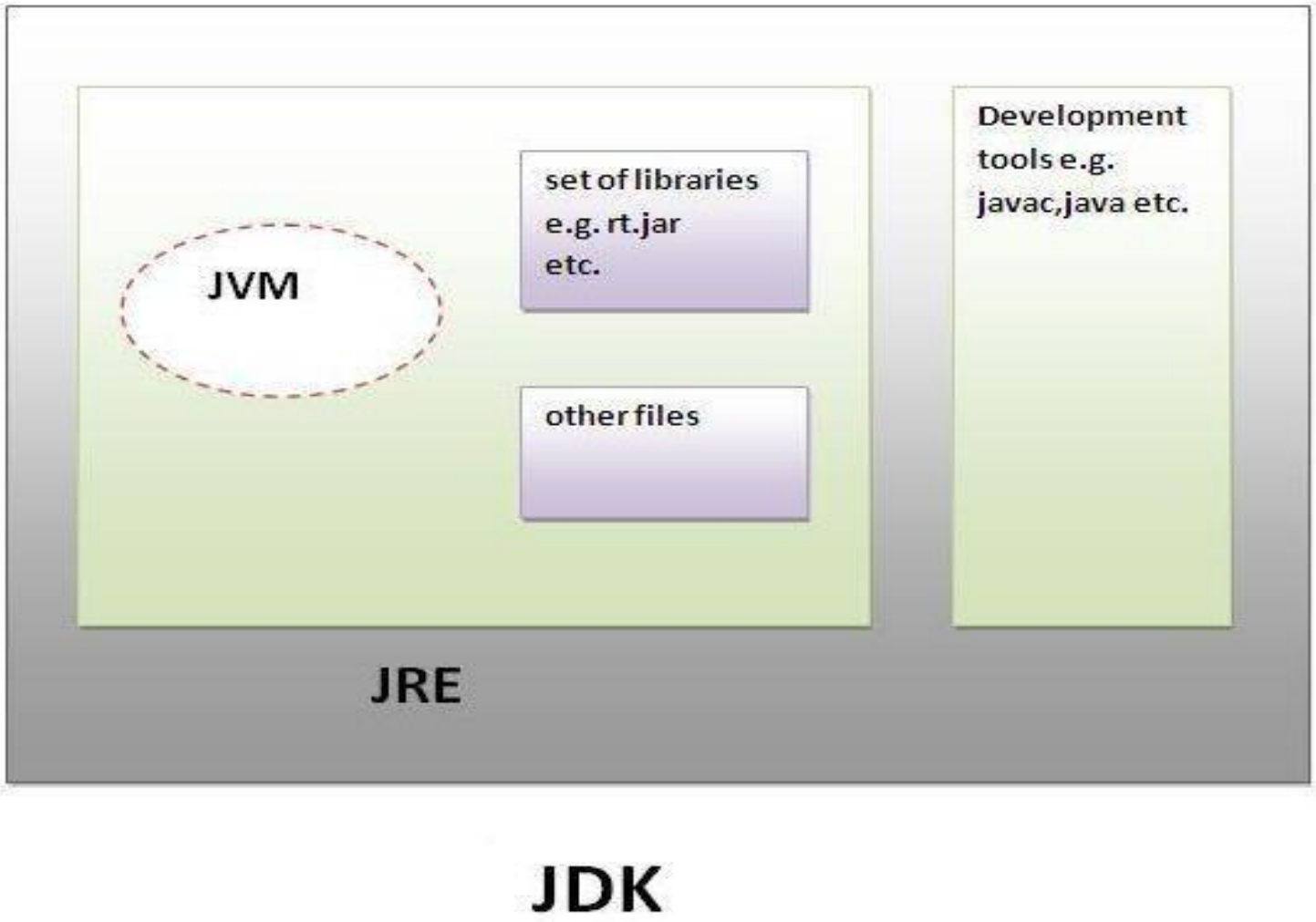
- In 1995, Java was first publicly released.
- In 1996, Java Development Kit (***JDK***) 1.0 was released.
- In 2002, JDK 1.4 (codename *Merlin*) was released, the most widely used version.
- In 2004, JDK 5.0 (codename *Tiger*) was released.
- The latest version of java is jdk 8.0.

Features of Java(12)




Class File
Windows JVM
Linux JVM
Mac/OS JVM
0
1
0
0
1
0
0
1
0
Windows
Linux
MAC/OS

Understanding JDK, JRE and JVM



Understanding JDK & JRE

JDK

- JDK is an acronym for *Java Development Kit*.
- It physically exists. It contains JRE and development tools.

JRE

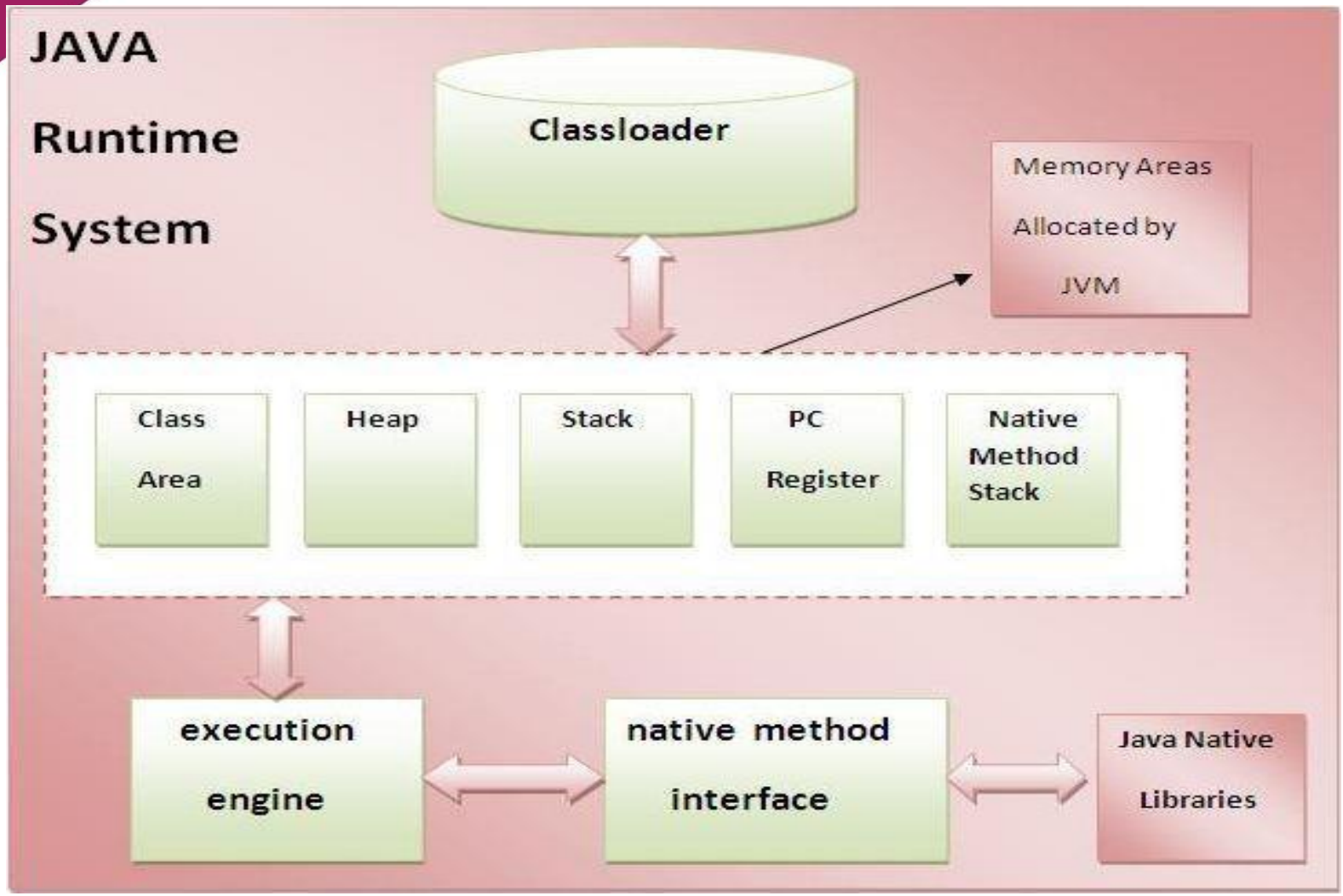
- JRE is an acronym for *Java Runtime Environment*.
- It is the implementation of JVM and used to provide runtime environment.
- It contains set of libraries and other files that JVM uses at runtime.

Understanding JVM

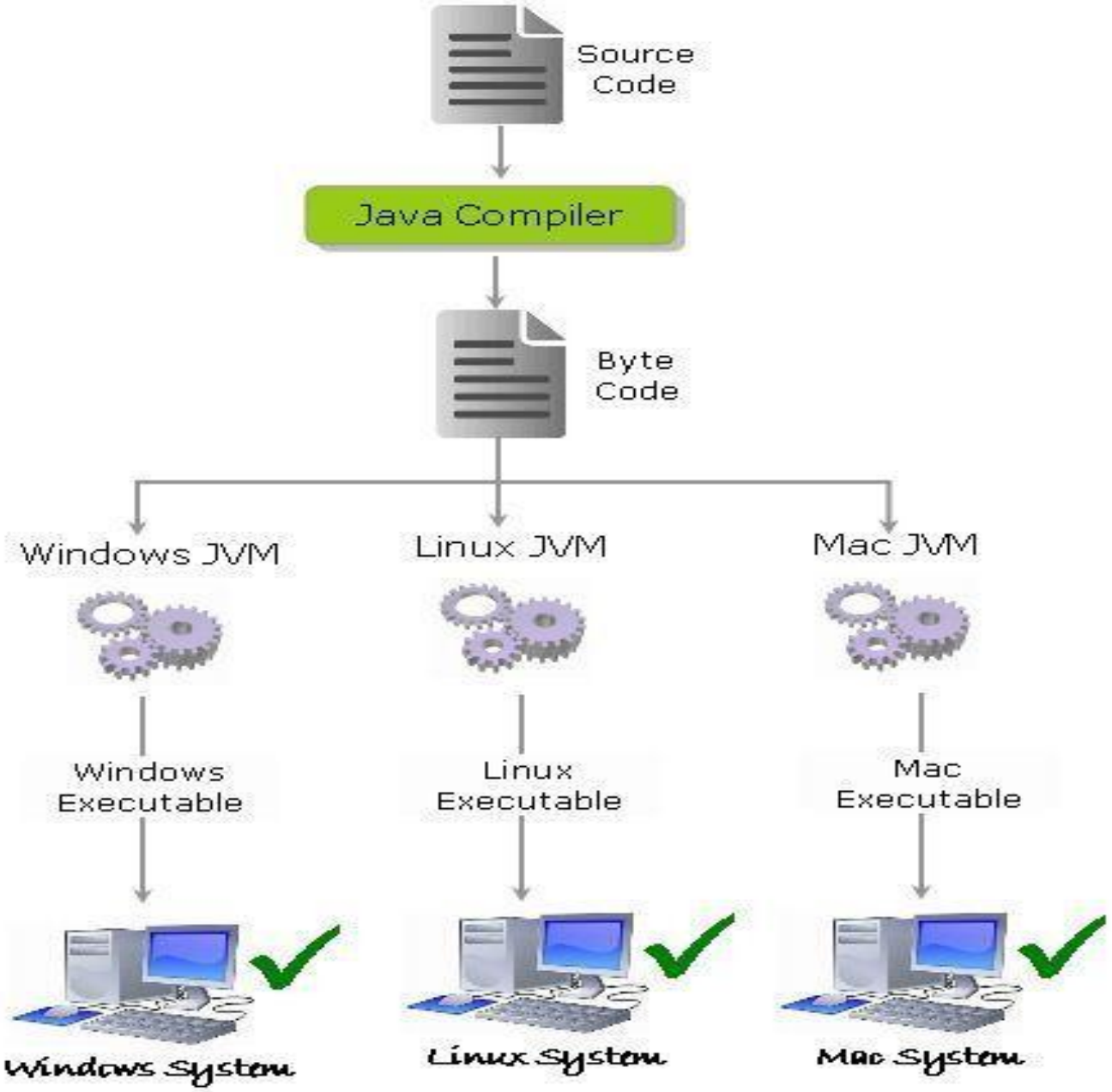
- JVM (Java Virtual Machine) is an abstract machine.
- It is a specification thatprovides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware andsoftware platforms.

- The JVM performs following main tasks:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

Internal Architecture



How Java is Platform-independent?


Source Code
Java Compiler
Byte Code
Windows JVM
Linux JVM
Mac JVM
Windows Executable
Linux Executable
Mac Executable
Windows System
Linux System
Mac System

How Java is Platform-independent?

- The source code (program) written in java is saved as a file with .java extension.

- The java compiler "javac" compiles the source code and produces the platform independent intermediate code called BYTE CODE. It is a highly optimized set of instructions designed to be executed by the JVM.

How Java is Platform-independent?

- The byte code is not native to any platform because java compiler doesn't interact with the local platform while generating byte code.
- It means that the Byte code generated on Windows is same as the byte code generated on Linux for the same java code.
- The Byte code generated by the compiler would be saved as a file with .class extension. As it is not generated for any platform, can't be directly executed on any CPU.

Class (Static) Variables

- Class variables are also known as static variables.
- Variable which are declared in a class, but outside a method, constructor or a block and qualified with 'static' keyword are known as class variables.
- *Used when we don't want to modify the value from object to object.*
- Only one copy of each class variable is created, regardless of how many objects are created from it.
- Static variables can be accessed by calling with the class name.

ClassName.VariableName

- Static variables are created with the start of execution of a program and destroyed when the program terminates.
- Default values are same as instance variables.
- A public static final variable behaves as a CONSTANT in Java.
- Static variables can be initialized using static block also.

Variable Initialization

Local variables must be initialized *explicitly by the programmer* as the default values are not assigned to them where as the instance variables and static variables are assigned *default values* if they are not assigned values at the time of declaration.

Brainstorming 1

What will be the output of the following Program?

```
class VariableDemo
{
  public static void main(String [] rk)
   {
   int x = 10;//if used without initialization will give error(int
   x)
   System.out.print(x);
   }
}
```

Brainstorming 2

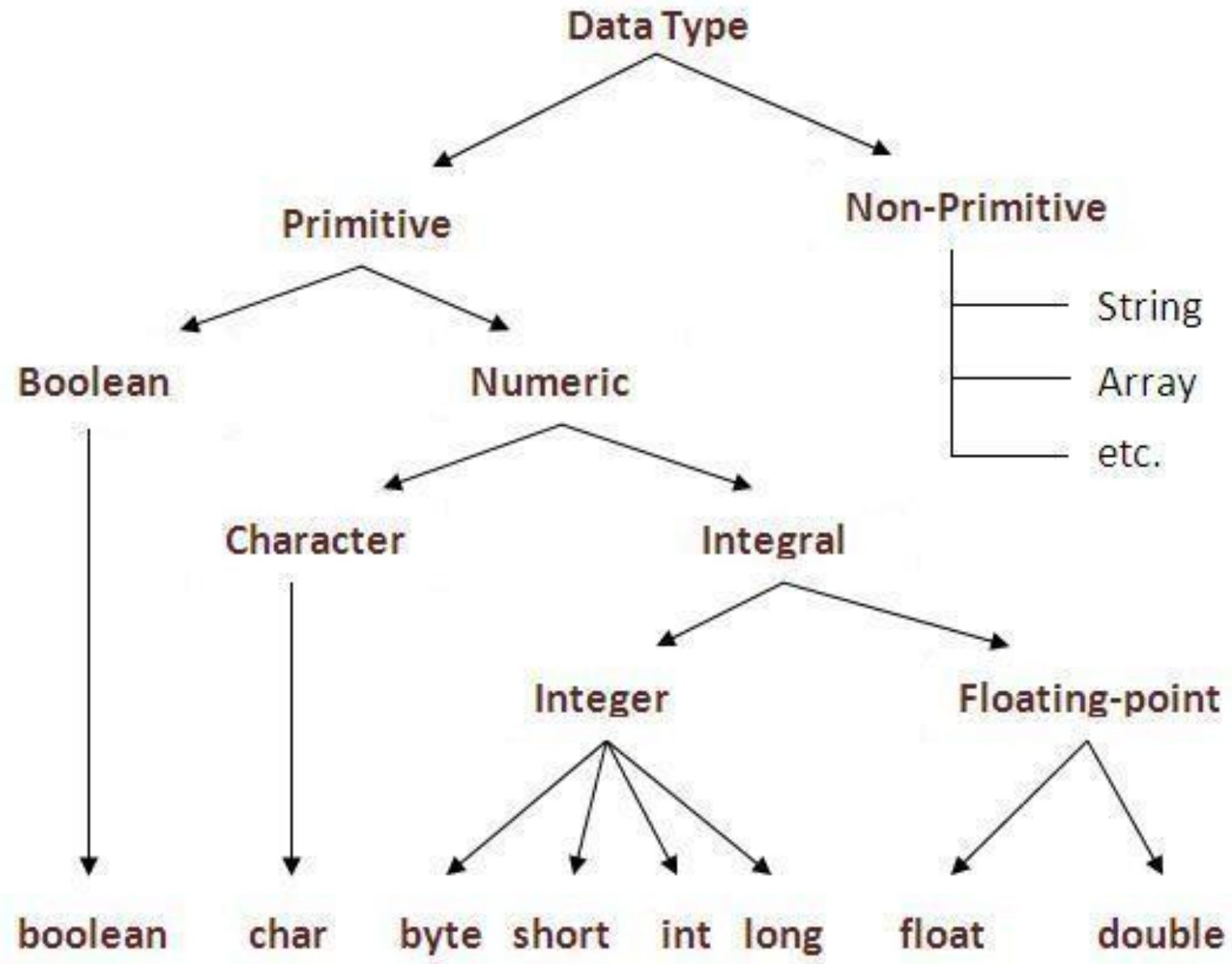
What will be the output of the following Program?

```
class VariableDemo
{
   static int x;
   public static void main(String [] rk)
    {
          System.out.print(x);
    }
}
```

Data Types

- Data types represent the different values to be stored in the variable.
- In Java, there are two types of data types:

1. Primitive data types
2. Non-primitive data types


Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double

Compound Assignments

- Arithmetic operators are combined with the simple assignment operator to create compound assignments.
- Compound assignment operators are +=, -=, *=, /+, %=
- For example, x+=1; and x=x+1; both increment the value of x by 1.

Relational Operators

- Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.
- It always returns boolean value i.e true or false.

Relational Operators

Operator	Description
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Comparison Operators

- The instanceof operator is used to compare an object to a specified type i.e. class or interface.

- It can be used to test if an object is an instance of a class or subclass, or an instance of a class that implements a particular interface.

Unary Operators

- The unary operators require only one operand.

Operator	Description
+	Unary plus operator; indicates positive value
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

Boolean Logical Operators

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

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

- The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	~ A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Bitwise Logical NOT

- Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

  00101010 becomes

  11010101

  after the NOT operator is applied.

Bitwise Logical AND

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

```
00101010    42
&
00001111            15
=
00001010    10
```

Bitwise Logical OR

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

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

Bitwise Logical XOR

- The XOR operator, **^, combines bits such that if exactly one operand is 1, then the result** is 1. Otherwise, the result is zero. The following example shows the effect of the **^.**

- Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

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

Logical Operators

- These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators.

- OR operator results in true when A is true , no matter what B is. Similarly, AND operator results in false when A is false, no matter what B is.

- If we use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

- For example

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

The above code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it.

The ? Operator

- Java includes a special ternary (three-way) operator, ?, that can replace certain types of if-then-else statements.
- The ? has this 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.
- Both expression2 and expression3 are required to return the same type, which can't be void.

int ratio = denom == 0 ? 0 : num / denom **;**

- When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark.

- If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression.

- If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression.

Left Shift Operator

- The left shift operator,<<, shifts all of the bits in a value to the left a specified number of times.

value << num

- Example:

01000001 65 << 2

00000100 4

Right Shift Operator

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

  value >> num

- It is also known as signed right shift.

- Example:

  00100011 35 >> 2

  00001000 8

- 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.

Unsigned Right Shift

- In these cases, to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.

- To accomplish this, we will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

- Example:
  - 11111111 11111111 11111111 11111111 –1 in binary as an int
  - >>>24
  - 00000000 00000000 00000000 11111111 255 in binary as an int

Operator Precedence

Highest			
( )	[]	.	
++	--	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	Op=		
Lowest			

SELECTION STATEMENTS

- Java supports two selection statements: **if** and **switch.**

if statement

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

- Each statement may be a single statement or a compound statement enclosed in curly braces (block).
- The condition is any expression that returns a **boolean value.**
- **The else** clause is optional.
- If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.
- In no case will both statements be executed.

Nested ifs

- A nested if is an if statement that is the target of another if or else.
- In nested ifs an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if (i == 10) {
        if (j < 20) a = b;
        if (k > 100) c = d;        // this if is
        else a = c;                // associated with this else
        }
else a = d;                        // this else refers to if(i == 10)
```

if-else-if Ladder

- A sequence of nested ifs is the if-else-if ladder.

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

- The if statements are executed from the top to down.

switch

- The switch statement is Java's multi-way branch statement.
- Provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- Provides a better alternative than a large series of **if-else-if statements.**

```
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 (i.e. constant not variable).
- Duplicate case values are not allowed.
- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the default statement is executed.
- 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.

- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

```
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;
                default:
                        System.out.println("i is greater than 2.");
                    }
                }
            }
```

ITERATION STATEMENTS (LOOPS)

Iteration Statements

- In Java, iteration statements (loops) are:
  - for
  - while, and
  - do-while

- A loop repeatedly executes thesame set of instructions until a termination condition is met.

While Loop

- While loop repeats a statementor block while its controlling expression is TRUE.
- The condition can be any Boolean expression.
- The body of the loop will be 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.

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

```
class While
{
    public static void main(String args[]) {
    int n = 10;
    char a = 'G';
    while(n > 0)
            {
              System.out.print(a);
              n--;
              a++;
            }
      }
}
```

- The body of the loop will not execute even once if the condition is false.

- The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

do-while

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

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

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

for Loop


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

- Initialization portion sets the value of loop control variable.
- Initialization expression is only executed once.
- Condition must be a Boolean expression. It usually tests the loop control variable against a target value.
- Iteration is an expression that increments or decrements the loop control variable.

The for loop operates as follows.

- When the loop first starts, the initialization portion of the loop is executed.
- Next, condition is evaluated. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed.

```
class ForTable
{
        public static void main(String args[])
         {
                int n;
                int x=5;
                for(n=1; n<=10; n++)
                  {
                   int p = x*n;
                   System.out.println(x+"*"+n +"="+ p);
                  }
        }
}
```

What will be the output?

```
class Loop
   {
      public static void main(String args[])
         {
            for(int i=0; i<5; i++);
                  System.out.println (i++);
         }
    }
```