CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Variables

- A variable is a storage location used to hold data values that can be modified and accessed during the execution of a program.
- Allowed Characters:
  - Letters: Uppercase (A-Z) and lowercase (a-z)
  - Digits: (0-9), but not as the first character
  - Underscores: (_)
  - Dollar Sign: ($), though its use is generally discouraged for regular variable names

```
public class Variables {
    public static void main(String[] args) {
        // Valid variable names
        // Valid: starts with a letter, only contains letters & digits
         int userAge = 70;
        // Valid: starts with an underscore, contains letters
         String _userName = “James Gosling";
        // Valid: contains a dollar sign, starts with a letter
         double annualSalary$ = 10000000.50;

         // Invalid variable names (cause errors)
        /*
         int 1stVariable = 100;        // Invalid: starts with a digit
         String user-age = "Alice";    // Invalid: contains a hyphen
         double @salary = 199.99;  // Invalid: contains @ symbol
         boolean user Name = true;  // Invalid: contains a space
        */
  }
}
```

- **Starting Character**: Variable names must start with a letter (A-Z or a-z) or an underscore (_)
- **Subsequent Characters**: After the first character, variable names can include letters, digits, underscores, and dollar signs.
- **Case Sensitivity**: Java is case-sensitive(**myVariable, MyVariable, and MYVARIABLE are different variables**)
- **Reserved Words**: Variable names cannot be Java reserved keywords.

Variable types

In Java programming the main types of variables are local, instance, class and parameters.

	Local Variables	Instance Variables or Non-static fields	Class Variables or Static Variables	Parameters
Scope:	Declared inside methods, constructors, or blocks.	Declared within a class but outside methods.	Declared with the static keyword within a class.	Declared in method or constructor definitions.
Lifetime:	Exist only during the execution of the method or block where they are declared.	Exist as long as the object of the class exists.	Exist for the duration of the program and are shared among all instances of the class.	Exist only during the execution of the method or constructor
Initialization:	Must be initialized before use.	Automatically initialized to default values if not explicitly initialized.	Automatically initialized to default values if not explicitly initialized.	Initialized with the values passed to the method or constructor when it is called.

```
public class Variable {
   // Class variable (static variable)
   static int classVariable;    // Default value: 0
   // Instance variable
   int instanceVariable; // Default value: 0
   // Constructor with parameter
   public Variable(int instanceVariable) {
     // Initialize instance variable via constructor parameter
      this.instanceVariable = instanceVariable;
   }
   public void method() {
      // Local variable
      int localVariable = 20; // must be initialized before use
      System.out.println("Class Variable: " + classVariable);
      System.out.println("Instance Variable:" + instanceVariable);
      System.out.println("Local Variable: " + localVariable);
   }
```

```
public static void main(String[] args) {
      // Create two instances of Variable
      Variable obj1 = new Variable(10);
      Variable obj2 = new Variable(20);
      // Modify the class variable
      Variable.classVariable = 100;
      // instance variables
      System.out.println("obj1");
      System.out.println("######");
      obj1.method();
      System.out.println("obj2");
      System.out.println("######");
      obj2.method();
      // class variable
      System.out.println("######");
      System.out.println("Class Variable:" +Variable.classVariable);
      // instance variables from each object
      System.out.println("obj1: " + obj1.instanceVariable);
      System.out.println("obj2: " + obj2.instanceVariable);
   }
}
```

```
obj1
######
Class Variable: 100
Instance Variable: 10
Local Variable: 20
obj2
######
Class Variable: 100
Instance Variable: 20
Local Variable: 20
######
Class Variable : 100
obj1: 10
obj2: 20
```

Operators

- Java provides a rich operator environment.
- Most of its operators can be divided into the following four groups:
  - Arithmetic
  - Bitwise
  - Relational
  - Logical

Arithmetic Operators

- The operands of the arithmetic operators must be of a numeric type.
- Cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.
- Integer Division: Result is an integer. Any fractional part is discarded.
- Floating-Point Division: Retain fractional results.

OPERATOR	Description
+ , -, *, /,%	Addition, subtraction, multiplication, division, modulus
+, -	Unary plus, unary minus
++, --	Increment, Decrement
+= , -=, *=, /=,%=	Addition assignment, subtraction assignment, multiplication assignment, division assignment, modulus assignment

The Bitwise Operators

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

OPERATOR	Description
~	Bitwise unary NOT
&, |, ^	Bitwise AND, OR, EXCLUSIVE OR
>>, <<, >>>	Shift Right, Shift Left, Shift Right with zero fill
&=, |=, ^=	Bitwise AND assignment, Bitwise OR assignment, Bitwise Exclusive OR assignment
>>=, <<=, >>>=	Shift Right Assignment, Shift Left Assignment, Shift Right with zero fill Assignment

```java
public class BitwiseOperators {
    public static void main(String[] args) {
        int a = 5; // binary: 0101
        int b = 3; // binary: 0011
        // Bitwise Operators
        System.out.println("Bitwise AND: " + (a & b)); // 1 (binary 0001)
        System.out.println("Bitwise OR: " + (a | b));  // 7 (binary 0111)
        System.out.println("Bitwise XOR: " + (a ^ b)); // 6 (binary 0110)
        System.out.println("Bitwise Complement: " + (~a)); // -6 (binary 11111010)
        // Bitwise Assignment Operators
        a &= b;
        System.out.println("Bitwise AND Assignment: " + a); // 1
        a = 5; // Resetting
        a |= b;
        System.out.println("Bitwise OR Assignment: " + a); // 7
        a = 5; // Resetting
        a ^= b;
        System.out.println("Bitwise XOR Assignment: " + a); // 6
        // Shift Operators
        System.out.println("Left Shift: " + (a << 1)); // 12 (binary 1100)
        System.out.println("Right Shift: " + (a >> 1)); // 3 (binary 0011)
        System.out.println("Unsigned Right Shift: " + (a >>> 1)); // 3 (binary 0011)
        // Negative number example for unsigned right shift
        int negNum = -5;
        System.out.println("Unsigned Right Shift of -5: " + (negNum >>> 1));    }}
```

```
Bitwise AND: 1
Bitwise OR: 7
Bitwise XOR: 6
Bitwise Complement: -6
Bitwise AND Assignment: 1
Bitwise OR Assignment: 7
Bitwise XOR Assignment: 6
Left Shift: 12
Right Shift: 3
Unsigned Right Shift: 3
Unsigned Right Shift of -5: 2147483642
```

```
class BitLogic {
      public static void main(String args[]) {
String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"          };
      int a = 3;
      int b = 6;
      int or = a | b;
      int and = a & b;
      int xor = a ^ b;
      int xnor = (~a & b) | (a & ~b);
      int not = ~a & 0x0f;

      System.out.println("       a = " + binary[a]);
      System.out.println("       b = " + binary[b]);
      System.out.println("     (a|b) or= " + binary[or]);
      System.out.println("     (a&b) and= " + binary[and]);
      System.out.println("     (a^b) xor = " + binary[xor]);
      System.out.println(" (~a&b|a&~b)xnor = " + binary[xnor]);
      System.out.println("      ~a = " + binary[not]);
   }
 }
```

```
a = 0011
 b = 0110
(a|b) or= 0111
(a&b) and= 0010
(a^b) xor = 0101
(~a&b|a&~b)xnor = 0101
 ~a = 1100
```

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. outcome of these operations is a **boolean** value.
- only integer, floating-point, and character operands may be compared to see which is greater or less than the other.

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

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

- The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero.
- In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators

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

OPERATOR	Description
!, &, |, ^	Logical Unary NOT, Logical AND, Logical OR, Logical XOR
&=, |=, ^=	Logical AND assignment, Logial OR assignment, Logical XOR assignment
||, &&	Short-circuit OR, short-circuit AND
==, !=	Equal To, Not Equal To
?:	Ternary(If Then Else)

Boolean Logical Operators

- The logical Boolean operators, **&**, |, and ^, operate on **boolean** values The logical **!** operator inverts the Boolean state:

!true==false

!false == true.

```
// Demonstrate the boolean logical operators.
class BooleanLogic {
   public static void main(String args[]) {
      boolean a = true;
      boolean b = false;
      boolean c = a | b;
      boolean d = a & b;
      boolean e = a ^ b;
      boolean f = (!a & b) | (a & !b);
      boolean g = !a;
      System.out.println("       a = " + a);
      System.out.println("       b = " + b);
      System.out.println("     a|b = " + c);
      System.out.println("     a&b = " + d);
      System.out.println("     a^b = " + e);
      System.out.println("!a&b|a&!b = " + f);
      System.out.println("      !a = " + g);
   }
}
```

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

Short-Circuit Logical Operators

Short-Circuit OR (||):

- Evaluates the right-hand operand only if the left-hand operand is false
- Prevent unnecessary computations or avoid errors (e.g., check if an object is null before accessing its methods).

Short-Circuit AND (&&):

- Evaluates the right-hand operand only if the left-hand operand is true.
- Avoid operations that could cause errors or be inefficient if the result is already determined by the left-hand operand.

Advantages of Short-Circuit Operators:

Efficiency: Reduces computational overhead by avoiding unnecessary evaluations.

Error Prevention: Avoids runtime errors or exceptions (e.g., NullPointerException) by ensuring certain conditions are met before executing potentially risky code.

```
public class ShortCircuitEffects {
    private static int globalCounter = 0;
    public static void main(String[] args) {
        boolean condition1 = false;
        boolean condition2 = true;
        // Using short-circuit AND
        if (condition1 && performSideEffect()) {
            System.out.println("Condition met with AND");
        } else {
            System.out.println("Condition not met with AND");
        }
        // Using short-circuit OR
        if (condition2 || performSideEffect()) {
            System.out.println("Condition met with OR");
        } else {
            System.out.println("Condition not met with OR");
        }
        System.out.println("Global Counter: " + globalCounter);
    }
    private static boolean performSideEffect() {
        globalCounter++;
        System.out.println("Side effect executed");
        return true;
    }}
```

```
Condition not met with AND
Condition met with OR
Global Counter: 1
```

```
public class ShortCircuitExample {
    private static int globalCounter = 0;

    public static void main(String[] args) {
        boolean conditionA = false;
        boolean conditionB = false;

        // Using short-circuit OR (||)
        if (conditionA || performSideEffect()) {
            System.out.println("Condition met with OR");
        } else {
            System.out.println("Condition not met with OR");
        }

        // Print the final value of globalCounter
        System.out.println("Global Counter: " + globalCounter);
    }

    private static boolean performSideEffect() {
        // Side effect: modifying a global variable
        globalCounter++;
        System.out.println("Side effect executed");
        return true;
    } }
```


Avoids Unintended State Changes

If the right-hand side of a logical expression (&& or ||) has side effects, and the left-hand side of the expression already determines the result, the right-hand side might not be executed.

This avoids unintended changes or operations

```
Side effect executed
Condition met with OR
Global Counter: 1
```

If conditionA is true, performSideEffect() is not called due to short-circuit evaluation.
This prevents the global variable globalCounter from being incremented if it's not necessary

```
public class ShortCircuitExample {
    public static void main(String[] args) {
        boolean isValid = true;

        // Using short-circuit AND (&&)
        if (isValid && expensiveComputation()) {
            System.out.println("Condition met with AND");
        } else {
            System.out.println("Condition not met with AND");
        }
    }

    private static boolean expensiveComputation() {
        // Simulate an expensive computation
        System.out.println("Expensive computation started...");
        try {
            Thread.sleep(1000); // 1 second delay
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Expensive computation finished.");
        return false;
    }
}
```

Avoids expensive computation

Short-circuit evaluation can enhance performance by avoiding expensive or time-consuming operations when the result of the expression is already known.

short-circuit AND (&&) avoid an expensive computation when it is not necessary.

If isValid were false, the expensiveComputation() method would not be called at all, which is a key benefit of using short-circuit logical operators; the output is

```
Condition not met with AND
```

```
Expensive computation started...
Expensive computation finished.
Condition not met with AND
```

```
public class ShortCircuitExample {
    public static void main(String[] args) {
        String str = null;

 // Using short-circuit AND (&&) to avoid NullPointerException
        if (str != null && str.length() > 5) {
            System.out.println("String is longer than 5 characters");
        } else {
            System.out.println("String is null or not longer than 5
characters");
        }
    }
}
```

```
String is null or not longer than 5 characters
```

```
public class ShortCircuitExample {
    public static void main(String[] args) {
        String str = null;

        // Using short-circuit AND (&&) to avoid
NullPointerException
        if (str != null || str.length() > 5) {
            System.out.println("String is longer than 5 characters");
        } else {
            System.out.println("String is null or not longer than 5
characters");
        }
    }
}
```

```
Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"String.length()" because "str" is null
                at
ShortCircuitExample.main(ShortCircuitExample.java:10)
```

Avoid Exceptions:
Short-circuit operators can prevent exceptions by ensuring that the second operand is only evaluated if necessary.

The Assignment Operator

- The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:
- *var = expression*;
- Here, the type of *var* must be compatible with the type of *expression*.
- int x, y, z;

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

The ? Operator

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

- This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

condition ? expression1 : expression2

- condition: An expression that evaluates to a boolean value.
- expression1: Evaluated and returned if condition is true.
- expression2: Evaluated and returned if condition is false

- The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same (or compatible) type, which can't be void.

- Here is an example of the way that the ? is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

The ?: Operator

```
public class TernaryExample {
    public static void main(String[] args) {
        int i1 = 10;  // Example 1
        int i2 = -10; // Example 2

        int absValue1 = (i1 < 0) ? -i1 : i1;
        int absValue2 = (i2 < 0) ? -i2 : i2;

        System.out.println("Absolute value of " + i1 + " is " + absValue1);
        System.out.println("Absolute value of " + i2 + " is " + absValue2);
    }
}
```

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

- In Java, operators have a specific order of precedence, determining how expressions are evaluated.
- The order of precedence from highest to lowest is as follows.
- Although [ ], ( ), and . are technically separators, they also function as operators with the highest precedence when used for array access, method calls, and field access.
- Binary Operations: Evaluated from left to right.
- Assignment Operators: Evaluated from right to left

Operator Precedence

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	-(unary)	(Type-cast)
*	/	%				
+	-					
>>	<<	>>>				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	op= (+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, and >>>=)					
Lowest						

```
public class OperatorPrecedence {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int c = 2;
            // Postfix and prefix operators
        System.out.println(a++ + " " + a);
            // Prefix increment
        System.out.println(++b + " " + b);
            // Unary operators
        System.out.println("\nUnary operators:");
        System.out.println("Bitwise complement of 5: " + ~a);
        System.out.println("Logical NOT of true: " + !true);
        System.out.println("Unary plus: " + +a);
        System.out.println("Unary minus: " + -a);
            // Type casting
        System.out.println("\nType casting:");
        System.out.println("Int to double: " + (double)a);
        System.out.println("\nMultiplication, division, and modulus:");
         // Multiplication and division have same precedence
        System.out.println("a * b / c = " + (a * b / c));
        // Parentheses change the order
        System.out.println("a * (b / c) = " + (a * (b / c)));
        // Modulus has same precedence as multiplication
         System.out.println("a * b % c = " + (a * b % c));
            // Addition and subtraction
            System.out.println("\nAddition and subtraction:");
          // Left to right evaluation
            System.out.println("a + b - c = " + (a + b - c));
           // precedence across rows
            System.out.println("\n different precedence levels:");
        // Prefix increment, then multiplication, then addition
            System.out.println("++a * b + c = " + (++a * b + c));
        // Multiplication before addition
            System.out.println("a + b * c = " + (a + b * c));
        // Parentheses change the order
            System.out.println("(a + b) * c = " + ((a + b) * c));
        }
}
```

```
5 6
11 11

Unary operators:
Bitwise complement of 5: -7
Logical NOT of true: false
Unary plus: 6
Unary minus: -6

Type casting:
Int to double: 6.0

Multiplication, division, and modulus:
a * (b / c) = 30
a * b % c = 0

Addition and subtraction:
a + b - c = 15

 different precedence levels:
++a * b + c = 79
a + b * c = 29
(a + b) * c = 36
```

```
public class OperatorPrecedence {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int c = 2;
            // Postfix and prefix operators
        System.out.println(a++ + " " + a);
              // Prefix increment
        System.out.println(++b + " " + b);
            // Unary operators
        System.out.println("\nUnary operators:");
        System.out.println("Bitwise complement of 5: " + ~a);
        System.out.println("Logical NOT of true: " + !true);
        System.out.println("Unary plus: " + +a);
        System.out.println("Unary minus: " + -a);
            // Type casting
        System.out.println("\nType casting:");
        System.out.println("Int to double: " + (double)a);
```

```
 5 6
11 11

Unary operators:
Bitwise complement of 5: -7
Logical NOT of true: false
Unary plus: 6
Unary minus: -6

Type casting:
Int to double: 6.0
```

```
Multiplication, division, and modulus:
a * (b / c) = 30
a * b % c = 0

Addition and subtraction:
a + b - c = 15

 different precedence levels:
++a * b + c = 79
a + b * c = 29
(a + b) * c = 36
```

```
 // Multiplication, division, and modulus
 System.out.println("\nMultiplication, division, and modulus:");
 // Multiplication and division have same precedence
System.out.println("a * b / c = " + (a * b / c));
// Parentheses change the order
System.out.println("a * (b / c) = " + (a * (b / c)));
// Modulus has same precedence as multiplication
  System.out.println("a * b % c = " + (a * b % c));
        // Addition and subtraction
       System.out.println("\nAddition and subtraction:");
   // Left to right evaluation
       System.out.println("a + b - c = " + (a + b - c));
    // Demonstrating precedence across rows
       System.out.println("\n different precedence levels:");
// Prefix increment, then multiplication, then addition
       System.out.println("++a * b + c = " + (++a * b + c));
// Multiplication before addition
       System.out.println("a + b * c = " + (a + b * c));
// Parentheses change the order
       System.out.println("(a + b) * c = " + ((a + b) * c));
    }
}
```

Control statements

- To cause the flow of execution to advance and branch based on changes to the state of a program.
- Java’s program control statements can be put into the following categories:
  - Selection
  - Iteration
  - Jump

Control statements

- Selection statements:
  - To choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Iteration statements:
  - To enable program execution to repeat one or more
- Jump statements:
  - To allow your program to execute in a nonlinear fashion.

Java’s Selection Statements

- To control the flow of your program’s execution based upon conditions known only during run time.
- Java supports two selection statements:
  - if and switch.

Java's Selection Statements

- If statement
  - The if statement is Java's conditional branch statement.
  - Used to route program execution through two different paths.

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

- Here, each statement may be a single statement, or a compound statement enclosed in curly braces (that is, a block).
- The condition is any expression that returns a boolean value.
- The else clause is optional.
- The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.
- The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.
- If there is no final **else** and all other conditions are **false**, then no action will take place.

Nested If

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

Selection statement: If statement

```
class ConditionalExamples {
    public static void main(String args[]) {
        // First example
        int bytesAvailable = 10; // Example value
        int n = 5; // Example value
        if (bytesAvailable > 0) {
            ProcessData();
            bytesAvailable -= n;
        } else {
            waitForMoreData();
            bytesAvailable = n;
        }
        // Second example
        boolean dataAvailable = true;
        if (dataAvailable)
            ProcessData();
        else
            waitForMoreData();

        // Third example
        int month = 4; // April
        String season;
        if (month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if (month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if (month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if (month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");     }
    private static void ProcessData() {
        System.out.println("Processing data...");    }
     private static void waitForMoreData() {
        System.out.println("Waiting for more data...");
     }
}
```

Selection statement: Switch

- The switch statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- A better alternative than a large series of if-else-if statements.
- Expression must resolve to type byte, short, int, char, String or an enumeration.
- Duplicate case values are not allowed. The type of each value must be compatible with the type of expression.
- The expression is evaluated once.
- The break statement is used to exit the switch statement. Without break, the program continues to the next case.
- If no cases match, the default block is executed (if it exists).
- Switch can handle byte, short, char, int, String, enum primitive data types and Wrapper classes (Integer, Byte, Short, Character)

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

```
class SwitchExample {
   public static void main(String args[]) {
            byte b = 1; short s = 100; char c = 'A'; int i = 10;
            switch (b) {// Example with byte
               case 1:
                  System.out.println("Byte value is 1");
                  break;
               default:
                  System.out.println("Byte value is neither 1 nor 2");
                  break;                }
            switch (s) {// Example with short
               case 100:
                  System.out.println("Short value is 100");
                  break;
               default:
                  System.out.println("Short value is neither 100 nor 200");
                  break;                }
           switch (c) {// Example with char
               case 'A':
                  System.out.println("Char value is A");
                  break;
               default:
                  System.out.println("Char value is neither A nor B");
                  break;
            }
```

```
switch (i) {
            // Example with int
               case 10:
                  System.out.println("Int value is 10");
                  break;
               default:
      System.out.println("Int value is neither 10 nor 20");
                  break;                }
 // Example with String
      String str = "hello";
            switch (str) {
               case "hello":
                  System.out.println("String is 'hello'");
                  break;
               case “java":
                  System.out.println("JamesGosling");
                  break;
               default:
 System.out.println("neither 'hello' nor 'JamesGosling'");
                  break;                }
```

```
// Example with enum
            Day day = Day.MONDAY;
            switch (day) {
                case MONDAY:
                    System.out.println("It's Monday");
                    break;
                case TUESDAY:
                    System.out.println("It's Tuesday");
                    break;
                default:
                    System.out.println("It's neither Monday nor Tuesday");
                    break;
            }
        }

        // Enum type for the days of the week
        enum Day {
            MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY, SUNDAY
        }
    }       }
```

```
Byte value is 1
Short value is 100
Char value is A
Int value is 10
String is 'hello'
It's Monday
```

case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch.

```
class NestedSwitch {
    public static void main(String[] args) {
        int mode = 2, option = 1;

        switch (mode) {
            case 1 -> switch (option) {
                case 1 -> System.out.println("Mode 1, Option 1");
                case 2 -> System.out.println("Mode 1, Option 2");
                case 3 -> System.out.println("Mode 1, Option 3");
            };
            case 2 -> switch (option) {
                case 1 -> System.out.println("Mode 2, Option 1");
                case 2 -> System.out.println("Mode 2, Option 2");
                case 3 -> System.out.println("Mode 2, Option 3");
            };
        }
    }
}
```

```
class NestedSwitch {
    public static void main(String[] args) {
        int mode = 2, option = 1;

        switch (mode) {
            case 1:
                switch (option) {
                    case 1: System.out.println("Mode 1, Option 1"); break;
                    case 2: System.out.println("Mode 1, Option 2"); break;
                    case 3: System.out.println("Mode 1, Option 3"); break;
                }
                break;
            case 2:
                switch (option) {
                    case 1: System.out.println("Mode 2, Option 1"); break;
                    case 2: System.out.println("Mode 2, Option 2"); break;
                    case 3: System.out.println("Mode 2, Option 3"); break;
                }
                break;
        }
    }
}
```

Selection statement: Switch

```
import java.util.Scanner;

public class SimpleCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        double num1, num2, result;
        char operation;
        System.out.println("Simple Calculator");
        System.out.println("Enter first number:");
        num1 = scanner.nextDouble();
        System.out.println("Enter an operation (+, -, *, /):");
        operation = scanner.next().charAt(0);
        System.out.println("Enter second number:");
        num2 = scanner.nextDouble();
        switch (operation) {
            case '+':
                result = num1 + num2;
                System.out.printf("%.2f + %.2f = %.2f", num1, num2, result);
                break;
            case '-':
                result = num1 - num2;
                System.out.printf("%.2f - %.2f = %.2f", num1, num2, result);
                break;
            case '*':
                result = num1 * num2;
                System.out.printf("%.2f * %.2f = %.2f", num1, num2, result);
                break;
            case '/':
                if (num2 != 0) {
                    result = num1 / num2;
                    System.out.printf("%.2f / %.2f = %.2f", num1, num2, result);
                } else {
                    System.out.println("Error: Division by zero!");                }
                break;
            default:
                System.out.println("Error: Invalid operation!");
        } } }
```

```
Simple Calculator
Enter first number:
10
Enter an operation (+, -, *, /):
#
Enter second number:
2
Error: Invalid operation!
```

```
Simple Calculator
Enter first number:
10
Enter an operation (+, -, *, /):
*
Enter second number:
2
10.00 * 2.00 = 20.00
```

Selection statement: Switch

- Three important features of the switch statement to note:
- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
- A switch statement is usually more efficient than a set of nested ifs in terms of execution speed and code clarity, particularly when there are many cases to evaluate

Selection statement: switch vs if

- To select among a large group of values, a **switch** statement will run much faster than the equivalent logic coded using a sequence of **if-else**s.
- The compiler can do this because it knows that the **case** constants are all the same type and simply must be compared for equality with the **switch** expression.
- The compiler has no such knowledge of a long list of **if** expressions.

Iteration Statements:

- Java’s iteration statements are
  - for,
  - while, and
  - do-while.
- These statements are commonly call loops.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

Iteration statement: While

- The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

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

- 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.
- The curly braces are unnecessary if only a single statement is being repeated.

```
class NoBody {
    public static void main(String[] args) {
        int i, j;

        i = 100;
        j = 200;

        // Find midpoint between i and j
        while (++i < --j); // no body in this loop

        System.out.println("Midpoint is " + i);
    }
}
```

```
Midpoint is 150
```

```
class SimpleWhileLoop {
    public static void main(String[] args) {
        int count = 1;  // Initialize the counter
         while (count <= 5) {
            System.out.println("Count is: " + count);
            count++;  // Increment the counter
        }
    }
}
```

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

Iteration statement: do-while

- 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.

- Java's loops, condition must be a Boolean expression.

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

```
import java.util.Scanner;
// validate user input

class DoWhileLoopExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int number;
       do {
            System.out.print("Please enter a number between 1 and 10: ");
            number = scanner.nextInt();

            if (number < 1 || number > 10) {
                System.out.println("Invalid input. Try again.");
            }
        } while (number < 1 || number > 10);

        System.out.println("You entered: " + number);
    }
}
```

```
Please enter a number between 1 and 10: 5
You entered: 5
```

```
Please enter a number between 1 and 10: 11
Invalid input. Try again.
```

Iteration statement: While vs do While

- While loop
  - The conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all.
- Do while loop
  - Execute the body of a loop at least once, even if the conditional expression is false to begin with.
  - To test the termination expression at the end of the loop rather than at the beginning.
  - The do-while loop always executes its body at least once.

Iteration statement:for

- There are two forms of the for loop.
  - for loop
    - Requires manual management of the index and loop bounds.
    - Indexing required
    - Read and Write access

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

  - for each loop
    - iterates directly over elements in arrays or collections
    - No indexing required
    - Read-only access

```
for (Type element : Array or collection) {
 //  body of the loop
 }
```

Iteration statement: for

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

- First, the initialization portion of the loop is executed only once when the loop starts.
- Next, Boolean expression 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.
- 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.

Iteration statement:for

- There are two forms of the for loop.
  - for loop

```
String[] javaDevelopers = {"James", "Patrick", "Mike"};
for (int i = 0; i < javaDevelopers.length; i++) {
System.out.println(javaDevelopers[i]);
}
```

  - for each loop

```
String[] javaDevelopers = {"James", "Patrick", "Mike"};
for (String name : javaDevelopers) {
System.out.println(name);
 }
```

Basic for Loop

```
class BasicForLoop {
    public static void main(String[] args) {
        // Print squares of numbers from 1 to 5
        for (int i = 1; i <= 5; i++) {
            System.out.println("Square of " + i + " is " +
(i * i));
        }
    }
}
```

```
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
```

For loop with variable declared inside:

```
class VariableInsideForLoop {
 public static void main(String[] args) {
    // Print Fibonacci sequence up to 100
    int prev = 0;
    System.out.print("Fibonacci sequence: ");
    for (int current = 1; current <= 100; ) {
                System.out.print(current + " ");
                 int next = prev + current;
                prev = current;
                current = next;
    }
    }
}
```

```
Fibonacci sequence: 1 1 2 3 5 8 13 21 34 55 89
```

For loop with boolean condition:

```
class BooleanConditionForLoop {
    public static void main(String[] args) {
       int sum = 10;
        boolean reachedTarget = false;
        int num ;
        for ( num=1; !reachedTarget; num++) {
            sum += num;
            if (sum > 100) {
                reachedTarget = true;
            }
        }
        System.out.println("Sum exceeded 100 after adding " + num+ " numbers");    }
}
```

Sum exceeded 100 after adding 14 numbers

For loop with some parts empty:

```
class EmptyPartsForLoop {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        int index = 0;

        for (; index < numbers.length;) {
            System.out.println("Element at index " + index + ": " +
numbers[index]);
            index++;
        }
    }
}
```

Element at index 0: 1
Element at index 1: 2
Element at index 2: 3
Element at index 3: 4
Element at index 4: 5

```
class EmptyPartsForLoop {
   public static void main(String[] args) {
        int[]  numbers = {1, 2, 3, 4, 5};
       int index = 0;
       Boolean complete=false;
       for (; !complete;) {
          System.out.println("Element at index " + index + ": " + numbers[index]);
          if(index==4) complete=true;
          index++;
    }
 }
}
```

```
Element at index 0: 1
Element at index 1: 2
Element at index 2: 3
Element at index 3: 4
Element at index 4: 5
```

An infinite loop. This loop will run forever because there is no condition under which it will terminate.

```
class InfiniteForLoop {
    public static void main(String[] args) {
        int counter = 0;
        for (;;) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop");
                break;
            }
        }
    }
}
```



```
class InfiniteWhileLoop {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop"
                break;
            }
        }
    }
}
```

An infinite loop. This loop will run forever because there is no condition under which it will terminate.

```
for (;;) {
    // code
    if (/* condition */) {
        break; // Exit loop based on a condition
    }
}
```

```
while (true) {
// code
  if (/* condition */) {
      break; // Exit loop based on a condition
  }
}
```

```
class InfiniteForLoop {
    public static void main(String[] args) {
        int counter = 0;
        for (;;) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop");
                break;
            }
        }
    }
}
```

```
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
Breaking out of the infinite loop
```

```
class InfiniteWhileLoop {
    public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("This is iteration " + (++counter));
            if (counter == 5) {
                System.out.println("Breaking out of the infinite loop");
                break;
            }
        }
    }
}
```

```
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
Breaking out of the infinite loop
```

For loop with multiple variables:
To allow two or more variables to control a for loop, Java permits you to include multiple statements in both the initialization and iteration portions of the for. Each statement is separated from the next by a comma.

```
class MultipleVariablesForLoop {
    public static void main(String[] args) {
        // Print a countdown with days and hours
        for (int days = 3, hours = 0; days >= 0; days--, hours += 6) {
            System.out.println("Time remaining: " + days + " days and " + hours + " hours");
        }
    }
}
```

```
Time remaining: 3 days and 0 hours
Time remaining: 2 days and 6 hours
Time remaining: 1 days and 12 hours
Time remaining: 0 days and 18 hours
```

Iteration statement: For-Each version of the for loop

- A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

```
//The general form of the for-each
for(type itr-var : collection)
statement-block
```

  - type specifies the type
  - itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.

Basic for-each loop:

```
// Basic for-each style loop
class ForEachExample {
    public static void main(String[] args) {
        String[] fruits = {"Apple", "Banana", "Cherry", "Date", "berry"};
        int totalLength = 0;
        // for-each style to display and sum the length of fruit names
        for (String fruit : fruits) {
            System.out.println("Fruit name: " + fruit);
            totalLength += fruit.length();
        }
        System.out.println("Total length of all fruit names: " + totalLength);
    }
}
```

```
Fruit name: Apple
Fruit name: Banana
Fruit name: Cherry
Fruit name: Date
Fruit name: berry
Total length of all fruit names: 26
```

For-each loop with break

```
class ForEachWithBreak {
    public static void main(String[] args) {
        double[] prices = {10.99, 5.49, 15.99, 20.00, 7.99, 30.50};
        double budget = 40.00;
        double totalSpent = 0;
        // for-each to sum prices until budget is exceeded
        for (double price : prices) {
            System.out.println("Checking item priced at: $" + price);
            if (totalSpent + price > budget) {
                break; // Stop if adding this item would exceed the budget
            }
            totalSpent += price;
        }
        System.out.println("Total spent within budget: $" + totalSpent);
    }
}
```

```
Checking item priced at: $10.99
Checking item priced at: $5.49
Checking item priced at: $15.99
Checking item priced at: $20.0
Total spent within budget: $32.47
```

For-each loop with 2D

```
class ForEachWith2D {
    public static void main(String[] args) {
        String[][] schedule = {
                {"Monday", "Math", "History"},
                {"Tuesday", "Science", "English"},
                {"Wednesday", "Art", "Music"}
        };

        // Use for-each to display the schedule
        for (String[] day : schedule) {
            for (String subject : day) {
                System.out.print(subject + "\t");
            }
            System.out.println();
        }
    }
}
```

```
Monday   Math      History
Tuesday  Science   English
Wednesday   Art    Music
```

Using type inference in for loops

```
class TypeInferenceInFor {
    public static void main(String[] args) {
        System.out.print("Powers of 2: ");
        for (var i = 1; i <= 128; i *= 2) {
            System.out.print(i + " ");
        }
        System.out.println();

       var temperatures = new double[] {98.6, 100.4, 97.3, 99.1, 98.8};
        System.out.print("Temperatures: ");
        for (var temp : temperatures) {
            System.out.printf("%.1f°F ", temp);
        }
        System.out.println();
    }
}
```

```
Powers of 2: 1 2 4 8 16 32 64 128
Temperatures: 98.6°F 100.4°F 97.3°F 99.1°F 98.8°F
```

Jump Statements

- Java supports three jump statements:
  - break,
  - continue, and
  - return.
- These statements transfer control to another part of your program.

Break

- Force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

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

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete..
```

Using break as labels

```
// Using break as a form of goto.
class Break {
   public static void main(String args[]) {
      boolean t = true;

      one: {
         two: {
            three: {
               System.out.println("I am executable in three block.");
               if(t) break three; // break out of three block
               System.out.println("I won't execute");
            }
            System.out.println("I am executable at block two");
            if(t) break two; // break out of two block
         }
         System.out.println("This is at block one.");
      }
   }
}
```

```
I am executable in three block.
I am executable at block two
This is at block one.
```

```
// Using break to exit from nested loops
class BreakLoop4 {
   public static void main(String args[]) {
      outer: for(int i=0; i<10; i++) {
         System.out.print("outer loop " + i + ": ");
         for(int j=0; j<20; j++) {
            if(j == 10) break outer; // exit inner and outer loops
            System.out.print(j + " ");
         }
         System.out.println("I won't execute");
      }
      System.out.println("Outer Loops complete.");
   }
}
```

```
outer loop 0: 0 1 2 3 4 5 6 7 8 9 Outer Loops complete.
```

Using break as labels

```
// This program contains an error.
class BreakWithErr {
    public static void main(String args[]) {
        one: for(int i=0; i<10; i++) {
            System.out.print("Pass " + i + ": ");
        }

        for(int j=0; j<10; j++) {
            if(j == 10) break one; // error- one is not in this scope
            System.out.print(j + " ");
        }
    }
}
```

java: undefined label: one

Using continue

- In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression.
- For all three loops, any intermediate code is bypassed.

```
// Usage of continue.
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

```
0 1
2 3
4 5
6 7
8 9
```

```
// Usage continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
            }
            System.out.print(" " + (i + j));
        }
        System.out.println();      }
}
```

```
0
1 2
2 3 4
3 4 5 6
4 5 6 7 8
5 6 7 8 9 10
6 7 8 9 10 11 12
7 8 9 10 11 12 13 14
8 9 10 11 12 13 14 15 16
9 10 11 12 13 14 15 16 17 18
```

jump statement:Return

- The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a

```
// Demonstrate return.
class Return {
   public static void main(String args[]) {
      boolean t = true;

      System.out.println(" I will execute.");

      if(t) return; // returning to caller

      System.out.println(“I won't execute.");
   }
}
```

I will execute.