Learning Java - A Foundational Journey

Session: 2

Variables, Data Types, and Operators

Objectives

- Explain variables and their purpose
- State the syntax of variable declaration
- Explain the rules and conventions for naming variables
- Explain data types
- Describe primitive and reference data types
- Describe escape sequence
- Describe format specifiers
- Identify and explain different type of operators
- Explain the concept of casting
- Explain implicit and explicit conversion

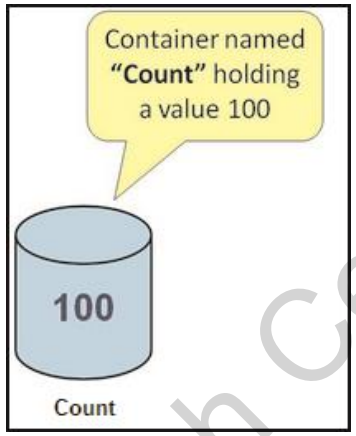
Introduction

- The core of any programming language is the way it stores and manipulates the data.
- The Java programming language can work with different types of data, such as number, character, boolean, and so on.
- To work with these types of data, Java programming language supports the concept of variables.
- A variable is like a container in the memory that holds the data used by the Java program.
- A variable is associated with a data type that defines the type of data that will be stored in the variable.
- Java is a strongly-typed language which means that any variable or an object created from a class must belong to its type and should store the same type of data.
- The compiler checks all expressions variables and parameters to ensure that they are compatible with their data types.

Variables 1-2

A variable is a location in the computer's memory which stores the data that is used in a Java program.

- Following figure depicts a variable that acts as a container and holds the data in it:



- Variables
  - Are used in a Java program to store data that changes during the execution of the program.
  - Are the basic units of storage in a Java program.
  - Can be declared to store values, such as names, addresses, and salary details.
  - Must be declared before they can be used in the program.
- A variable declaration begins with data type and is followed by variable name and a semicolon.

Variables 2-2

- The data type can be a primitive data type or a class.
- The syntax to declare a variable in a Java program is as follows:

Syntax

```
datatype variableName;
```

where,

`datatype`: Is a valid data type in Java.

`variableName`: Is a valid variable name.

- Following code snippet demonstrates how to declare variables in a Java program:

```
. . .
int rollNumber;
char gender;
. . .
```

- In the code, the statements declare an integer variable named **`rollNumber`**, and a character variable called **`gender`**.
- These variables will hold the type of data specified for each of them.

Assigning Value to a Variable 1-2

- Values can be assigned to variables by using the assignment operator (=).
- There are two ways to assign value to variables. These are as follows:

1. At the time of declaring a variable

- Following code snippet demonstrates the initialization of variables at the time of declaration:

```
...
int rollNumber = 101;
char gender = 'M';
...
```

- In the code, variable **rollNumber** is an integer variable, so it has been initialized with a numeric value **101**.
- Similarly, variable **gender** is a character variable and is initialized with a character '**M**'.
- Values assigned to the variables are called as literals.

Literals are constant values assigned to variables directly in the code without any computation.

Assigning Value to a Variable 2-2

After the variable declaration

- Following code snippet demonstrates the initialization of variables after they are declared:

```
int rollNumber; // Variable is declared
. . .
rollNumber = 101; //variable is initialized
. . .
```

- Here, the variable **`rollNumber`** is declared first and then, it has been initialized with the numeric literal **`101`**.
- Following code snippet shows the different ways for declaring and initializing variables in Java:

```
// Declares three integer variables x, y, and z
int x, y, z;
// Declares three integer variables, initializes a and c
int a = 5, b, c = 10;

// Declares a byte variable num and initializes its value to 20
byte num = 20;
// Declares the character variable c with value 'c'
char c = 'c';

// Stores value 10 in num1 and num2
int num1 = num2 = 10; //
```

- In the code, the declarations, `int x, y, z;` and `int a=5, b, c=10;` are examples of comma separated list of variables.
- The declaration `int num1 = num2 = 10;` assigns same value to more than one variable at the time of declaration.

Different Types of Variables

- Java programming language allows you to define different kind of variables that are categorized as follows:

Instance variables

- The state of an object is represented as fields or attributes or instance variables in the class definition.
- Each object created from a class will have its own copy of instance variables.

Static variables

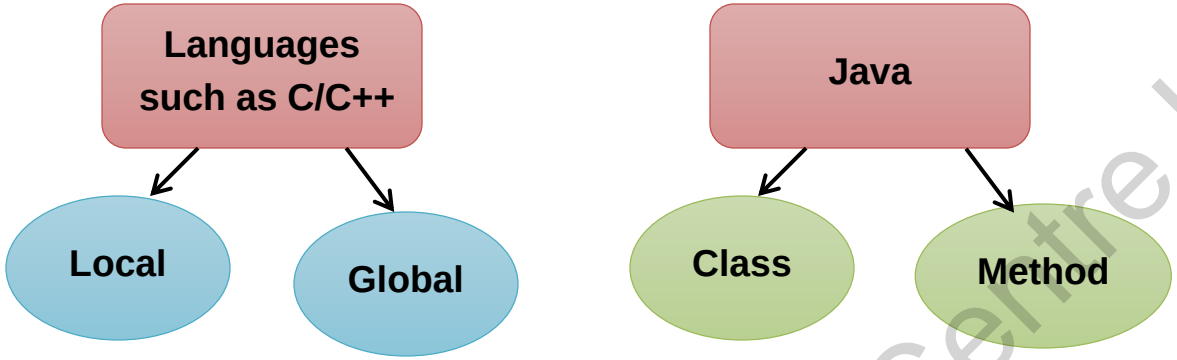
- These are also known as class variables.
- Only one copy of static variable is maintained in the memory that is shared by all the objects belonging to that class.
- These fields are declared using the `static` keyword.

Local variables

- The variables declared within the blocks or methods of a class are called local variables.
- A method represents the behavior of an object.
- The local variables are visible within those methods and are not accessible outside them.
- A method stores it temporary state in local variables.
- There is no special keyword available for declaring a local variable, hence, the position of declaration of the variable makes it local.

Scope of Variables

- In Java, variables can be declared within a class, method, or within any block.
- A scope determines the visibility of variables to other part of the program.



Class scope

- Variables declared within the class can be instance variables or static variables.
- The instance variables are owned by the objects of the class and their existence or scope depends upon the object creation.
- Static variables are shared between the objects and exists for the lifetime of a class.

Local Variable Type Inference

- The variables defined within the methods of a class are local variables.
- The lifetime of these variables depends on the execution of methods.
- This means memory is allocated for the variables when the method is invoked and destroyed when the method returns.
- After the variables are destroyed, they are no longer in existence.
- Methods parameters values passed to them during method invocation.
- The parameter variables are also treated as local variables which means their existence is till the method execution is completed.

Data Types

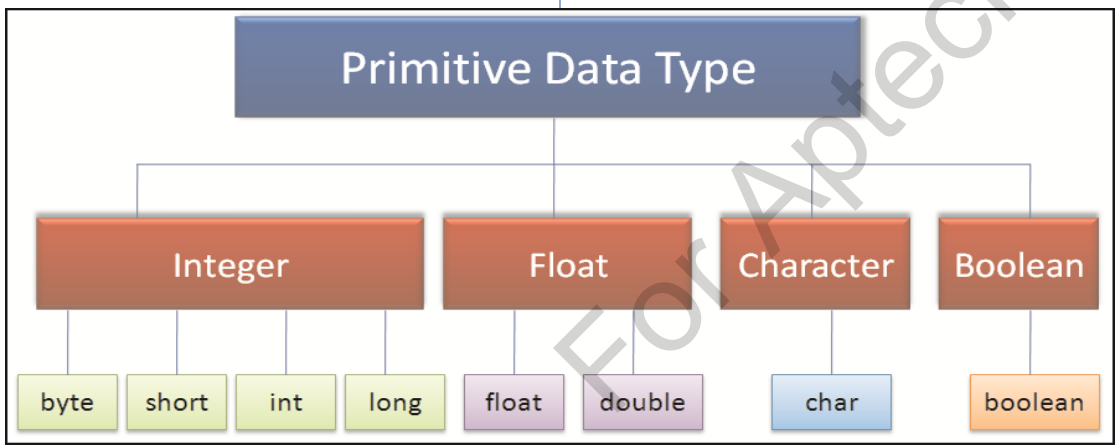
When you define a variable in Java, you must inform the compiler what kind of a variable it is.

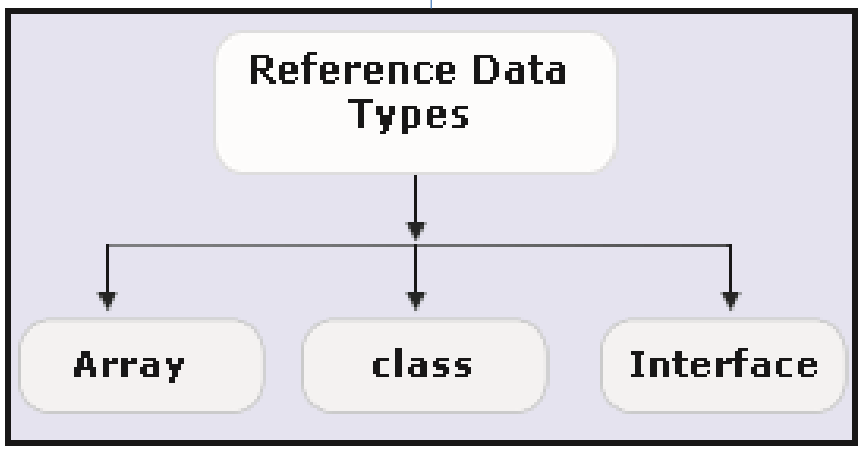
That is, whether it will be expected to store an integer, a character, or some other kind of data.

This information tells the compiler how much space to allocate in the memory depending on the data type of a variable.

Thus, the data types determine the type of data that can be stored in variables and the operation that can be performed on them.

In Java, data types fall under two categories:





Integer and Floating-point Types

- The integer data types supported by Java are `byte`, `short`, `int`, and `long`.
- These data types can store signed integer values.
- Signed integers are those integers, which are capable of representing positive as well as negative numbers, such as -40.
- Java does not provide support for unsigned integers.
- The floating-point data types supported by Java are `float` and `double`.
- These are also called real numbers, as they represent numbers with fractional precision.
- For example, calculation of a square root or PI value is represented with a fractional part.

Character Type and Boolean Type

- char data type belongs to this group and represents symbols in a character set like letters and numbers.
- char data type stores 16-bit Unicode character and its value ranges from 0 (‘\u0000’) to 65,535 (‘\uffff’).
- Unicode is a 16-bit character set, which contains all the characters commonly used in information processing.
- It is an attempt to consolidate the alphabets of the world’s various languages into a single and international character set.

- boolean data type represents true or false values.
- This data type is used to track true/false conditions.
- Its size is not defined precisely.
- Apart from primitive data types, Java programming language also supports strings.
- A string is a sequence of characters.
- Java does not provide any primitive data type for storing strings, instead provides a class String to create string variables.
- The String class is defined within the java.lang package in Java SE API.

String Type

- Following code snippet demonstrates the use of `String` class as primitive data type:

```
. . .
String str = "A String Data";
. . .
```

- The statement, `String str` creates an `String` object and is not of a primitive data type.

- When you enclose a string value within double quotes, the Java runtime environment automatically creates an object of `String` type.

- Also, once the `String` variable is created with a value '**`A String Data`**', it will remain constant and you cannot change the value of the variable within the program.

- However, initializing string variable with new value creates a new `String` object.

- This behavior of strings makes them as immutable objects.

Example

- Following code snippet demonstrates the use of different data types in Java:

```
public class EmployeeData {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {

      // Declares a variable of type integer
      int empNumber;
      //Declares a variable of type decimal
      float salary;
      // Declare and initialize a decimal variable
      double shareBalance = 456790.897;
      // Declare a variable of type character
      char gender = 'M';
      // Declare and initialize a variable of type boolean
      boolean ownVehicle = false;
      // Variables, empNumber and salary are initialized
      empNumber = 101;
      salary = 6789.50f;
      // Prints the value of the variables on the console
      System.out.println("Employee Number: " + empNumber);
      System.out.println("Salary: " + salary);
      System.out.println("Gender: " + gender);
      System.out.println("Share Balance: " + shareBalance);
      System.out.println("Owns vehicle: " + ownVehicle);
      }
      }
```

Reference Data Types

- In Java, objects and arrays are referred to as reference variables.
- Reference data type is an address of an object or an array created in memory.
- Following table lists and describes the three reference data types:

Data Type	Description
Array	It is a collection of several items of the same data type. For example, names of students in a class can be stored in an array.
Class	It is encapsulation of instance variables and instance methods.
Interface	It is a type of class in Java used to implement inheritance.

Literals 1-4

- A literal represents a fixed value assigned to a variable.
- It is represented directly in the code and does not require computation.
- Following figure shows some literals for primitive data types:

Integer	Float	Character	Boolean
50	35.7F	'C'	true

- A literal is used wherever a value of its type is allowed.
- However, there are several different types of literals as follows:

Integer Literals

- Integer literals are used to represent an int value, which in Java is a 32-bit integer value.
- Integers literals can be expressed as:

Decimal values have a base of 10 and consist of numbers from 0 through 9. For example, `int decNum = 56;`.

Hexadecimal values have a base of 16 and consist of numbers 0 through 9 and letters A through F. For example, `int hexNum = 0X1c;`.

Literals 2-4

Binary values have a base of 2 and consist of numbers 0 and 1. Java SE 7 supports binary literals. For example, `int binNum = 0b0010;`.

An integer literal can also be assigned to other integer types, such as `byte` or `long`.

- When a literal value is assigned to a `byte` or `short` variable, no error is generated, if the literal value is within the range of the target type.
- Integer numbers can be represented with an optional uppercase character ('`L`') or lowercase character ('`l`') at the end.
- This will inform the computer to treat that number as a long (64-bit) integer.

Floating-point Literals

- Floating-point literals represent decimal values with a fractional component.
- Floating-point literals have several parts.

Whole number component, for example 0, 1, 2,....., 9.

Decimal point, for example 4.90, 3.141, and so on.

Literals 3-4

Exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. For example, `e+208`, `7.436E6`, `23763E-05`, and so on.

Type suffix `D`, `d`, `F`, or `f`.

- Floating-point literals in Java default to double precision.
- A `float` literal is represented by `F` or `f` appended to the value, and a `double` literal is represented by `D` or `d`.

Boolean Literals

- Boolean literals are simple and have only two logical values - `true` and `false`.
- These values do not convert into any numerical representation.
- A `true` boolean literal in Java is not equal to one, nor does the `false` literal equals to zero.
- They can only be assigned to `boolean` variables or used in expressions with `boolean` operators.

Literals 4-4

Character Literals

- Character literals are enclosed in single quotes.
- All the visible ASCII characters can be directly enclosed within quotes, such as 'g', '$', and 'z'.
- Single characters that cannot be enclosed within single quotes are used with escape sequence.

Null Literals

- When an object is created, a certain amount of memory is allocated for that object.
- The starting address of the allocated memory is stored in an object variable, that is, a reference variable.
- However, at times, it is not desirable for the reference variable to refer that object.
- In such a case, the reference variable is assigned the literal value null. For example, `Car toyota = null;`.

String Literals

- String literals consist of sequence of characters enclosed in double quotes. For example, `"Welcome to Java", "Hello\nWorld"`.

Underscore Character in Numeric Literals

Java SE 7 allows you to add underscore characters (_) between the digits of a numeric literal.

The underscore character can be used only between the digits.

- In integral literals, underscore characters can be provided for telephone numbers, identification numbers, or part numbers, and so on.
- Similarly, for floating-point literals, underscores are used between large decimal values.
- Restrictions for using underscores in numeric literals are as follows:

A number cannot begin or end with an underscore.

In the floating-point literal, underscore cannot be placed adjacent to a decimal point.

Underscore cannot be placed before a suffix, `L` or `F`.

Underscore cannot be placed before or after the binary or hexadecimal identifiers, such as `b` or `x`.

Escape Sequences 1-2

An escape sequence is a special sequence of characters that is used to represent characters, which cannot be entered directly into a string.

- For example, to include tab spaces or a new line character in a line or to include characters which otherwise have a different notation in a Java program (\ or "), escape sequences are used.

An escape sequence begins with a backslash character (\), which indicates that the character (s) that follows should be treated in a special way.

The output displayed by Java can be formatted with the help of escape sequence characters.

- Following table lists escape sequence characters in Java:

Escape Sequence	Character Value
\b	Backspace character
\t	Horizontal Tab character
\n	New line character
\'	Single quote marks
\\	Backslash
\r	Carriage Return character
\"	Double quote marks
\f	Form feed
\xxx	Character corresponding to the octal value xxx, where xxx is between 000 and 0377
\uxxxx	Unicode character with encoding xxxx, where xxxx is one to four hexadecimal digits. Unicode escapes are distinct from the other escape types

Escape Sequences 2-2

- Following code snippet demonstrates the use of escape sequence characters:

```
public class EscapeSequence {
   /*
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // Uses tab and new line escape sequences
      System.out.println("Java \t Programming \n Language");
      // Prints Tom "Dick" Harry string
      System.out.println("Tom \"Dick\" Harry");
   }
}
```

- To represent a Unicode character, `\u` escape sequence can be used in a Java program.
- A Unicode character can be represented using hexadecimal or octal sequences.

Following code snippet demonstrates the Unicode characters in a Java program:

```
public class UnicodeSequence {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // Prints 'Hello' using hexadecimal escape sequence characters
      System.out.println("\u0048\u0065\u006C\u006C\u006F" + "!\n");
      // Prints 'Blake' using octal escape sequence character for 'a'
      System.out.println("Bl\141ke\"2007\" ");
   }
}
```

Constants and Enumerations 1-3

- Constants in Java are fixed values assigned to identifiers that are not modified throughout the execution of the code.
- In Java, the declaration of constant variables is prefixed with the `final` keyword.
- The syntax to initialize a constant variable is as follows:

Syntax

```
final data-type variable-name = value;
```

where,

`final`: Is a keyword and denotes that the variable is declared as a constant.

- Following code snippet demonstrates code that declares constant variables:

```
public class AreaOfCircle {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // Declares constant variable
    final double PI = 3.14159;
    double radius = 5.87;
    double area;
    // Calculates the value for the area variable
    area = PI * radius * radius;
    System.out.println("Area of the circle is: " + area);
  }
}
```

Constants and Enumerations 2-3

- Java SE 5.0 introduced enumerations.
- An enumeration is defined as a list that contains constants.
- Unlike C++, where enumeration was a list of named integer constants, in Java, enumeration is a class type.
- This means it can contain instance variables, methods, and constructors.
- The enumeration is created using the `enum` keyword.

- The syntax for declaring a method is as follows:

Syntax

```
enum enum-name {
constant1, constant2, . . . , constantN
}
```

Constants and Enumerations 3-3

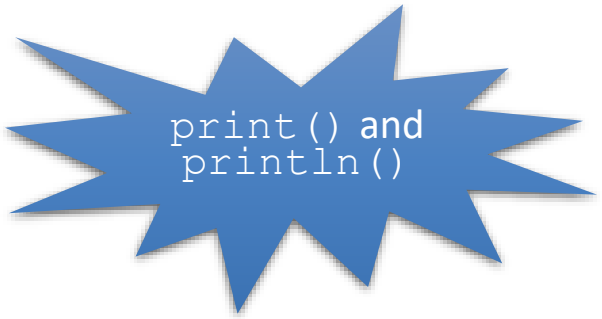
- Though, enumeration is a class in Java, you do not use `new` operator to instantiate it.
- Instead, declare a variable of type enumeration to use it in the Java program.
- This is similar to using primitive data types.
- The enumeration is mostly used with decision-making constructs, such as switch-case statement.

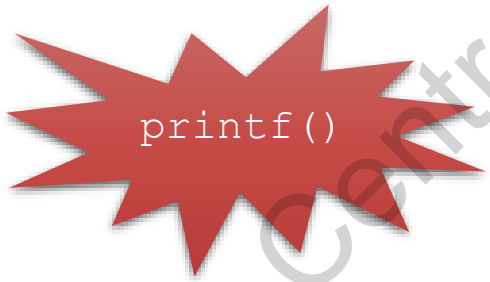
Following code snippet demonstrates the declaration of enumeration in a Java program:

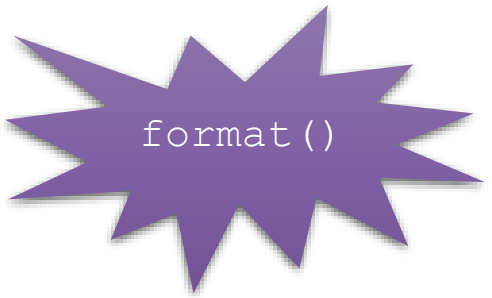
```
public class EnumDirection {
   /**
    * Declares an enumeration
    */
   enum Direction {
      East, West, North, South
   }

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // Declares a variable of type Direction
      Direction direction;

      // Instantiate the enum Direction
      direction = Direction.East;

      // Prints the value of enum
      System.out.println("Value: " + direction);
   }
}
```

Formatted Output and Input

- Whenever an output is to be displayed on the screen, it needs to be formatted.
- Formatting can be done using three ways that are as follows:







- These methods behave in a similar manner.
- The `format()` method uses the `java.util.Formatter` class to do the formatting work.

‘print()’ and ‘println()’ Methods

- These methods convert all the data to strings and display it as a single value.
- The method uses appropriate `toString()` method for conversion of the values.
- These methods can also be used to print mixture combination of strings and numeric values as strings on the standard output.

Following code snippet demonstrates use of print() and println() methods:

```
public class DisplaySum {
  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    int num1 = 5;
    int num2 = 10;
    int sum = num1 + num2;
    System.out.print("The sum of ");
    System.out.print(num1);
    System.out.print(" and ");
    System.out.print(num2);
    System.out.print(" is ");
    System.out.print(sum);
    System.out.println(".");
    int num3 = 2;
    sum = num1 + num2 + num3;
    System.out.println("The sum of " + num1 + ", " +
    num2 + " and " + num3 + " is " + sum + ".");
  }
}
```

‘printf()’ Method

- The `printf()` method introduced in J2SE 5.0 can be used to format the numerical output to the console.

- Following code snippet demonstrates the use of `printf()` methods to format the output:

```
public class FormatSpecifier {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int i = 55 / 22;
      // Decimal integer
      System.out.printf("55/22 = %d %n", i);
      // Pad with zeros
      double q = 1.0 / 2.0;
      System.out.printf("1.0/2.0 = %09.3f %n", q);
      // Scientific notation
      q = 5000.0 / 3.0;
      System.out.printf("5000/3.0 = %7.2e %n", q);
      // Negative infinity
      q = -10.0 / 0.0;
      System.out.printf("-10.0/0.0 = %7.2e %n", q);
      // Multiple arguments, PI value, E-base of natural logarithm
      System.out.printf("pi = %5.3f, e = %5.4f %n", Math.PI, Math.E);
   }
}
```

‘format()’ Method 1-2

- This method formats multiple arguments based on a format string.
- The format string consists of the normal string literal information associated with format specifiers and an argument list.
- The syntax of a format specifier is as follows:

Syntax

```
%[arg_index$] [flags] [width] [.precision] conversion character
```

where,

`arg_index`: Is an integer followed by a `$` symbol. The integer indicates that the argument should be printed in the mentioned position.

`flags`: Is a set of characters that format the output result. There are different flags available in Java.

Flag	Description
“-”	Left justify the argument
“+”	Include a sign (+ or -) with this argument
“0”	Pad this argument with zeros
“,”	Use locale-specific grouping separators
“(“	Enclose negative numbers in parenthesis

‘format()’ Method 2-2

- The values within ‘[]’ are optional.
- The only required elements of format specifier are the % and a conversion character.
- Following code snippet demonstrates the `format()` method:

```
public class VariableScope {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int num = 2;
      double result = num * num;
      System.out.format("The square root of %d is %f.%n", num, result);
   }
}
```

Formatted Input 1-2

- The `Scanner` class allows the user to read or accept values of various data types from the keyboard.
- It breaks the input into tokens and translates individual tokens depending on their data type.
- To use the `Scanner` class, pass the `InputStream` object to the constructor as follows:

```
Scanner input = new Scanner(System.in);
```

- Here, input is an object of `Scanner` class and `System.in` is an input stream object.
- Following table lists different methods of the `Scanner` class that can be used to accept numerical values from the user:

Method	Description
nextByte()	Returns the next token as a byte value
nextInt()	Returns the next token as an int value
nextLong()	Returns the next token as a long value
nextFloat()	Returns the next token as a float value
nextDouble()	Returns the next token as a double value

Formatted Input 2-2

- Following code snippet demonstrates the `Scanner` class methods and how they can be used to accept values from the user:

```
public class FormattedInput {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {

      // Creates an object and passes the inputstream object
      Scanner s = new Scanner(System.in);
      System.out.println("Enter a number:");
      // Accepts integer value from the user
      int intValue = s.nextInt();
      System.out.println("Enter a decimal number:");
      // Accepts integer value from the user
      float floatValue = s.nextFloat();
      System.out.println("Enter a String value");
      // Accepts String value from the user
      String strValue = s.next();
      System.out.println("Values entered are: ");
      System.out.println(intValue + " " + floatValue + " " + strValue);
   }
}
```

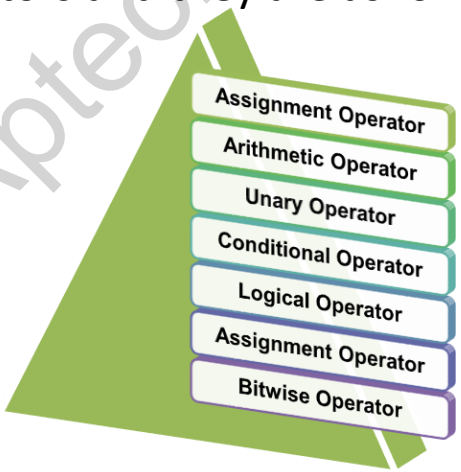
Operators

Operators are set of symbols used to indicate the kind of operation to be performed on data.

- Consider the expression: `Z = X + Y;`
- Here,

`+`	• Is called the Operator and the operation performed is addition.
`X and Y`	• The two variables **`X`** and **`Y`**, on which addition is performed, are called as Operands.
`Z = X + Y`	• A combination of both the operator and the operands, is known as an Expression.

- Java provides several categories of operators and they are as follows:



Assignment Operators

The basic assignment operator is a single equal to sign, '='.

This operator is used to assign the value on its right to the operand on its left.

Assigning values to more than one variable can be done at a time.

In other words, it allows you to create a chain of assignments.

- Consider the following statements:

```
int balance = 3456;
char gender = 'M';
```

- The value **3456** and '**M**' are assigned to the variables, **balance** and **gender**.
- In addition to the basic assignment operator, there are combined operators that allow you to use a value in an expression, and then, set its value to the result of that expression.

```
X = 3;
X += 5;
```

- The second statement stores the value **8**, the meaning of the statement is that **X = X + 5**.
- Following code snippet demonstrates use of assignment operators:

```
...
x = 10; // Assigns the value 10 to variable x
x += 5; // Increments the value of x by 5
x -= 5; // Decrements the value of x by 5
x *= 5; // Multiplies the value of x by 5
x /= 2; // Divides the value of x by 2
x %= 2; // Divides the value of x by 2 and the remainder is returned
```

Arithmetic Operators

- Arithmetic operators manipulate numeric data and perform common arithmetic operations on the data.
- Operands of the arithmetic operators must be of numeric type.
- Boolean operands cannot be used, but character operands are allowed.
- The operators mentioned here are binary in nature that is, these operate on two operands, such as **X+Y**. Here, + is a binary operator operating on **X** and **Y**.
- Following table lists the arithmetic operators:

Operator	Description
+	Addition - Returns the sum of the operands
-	Subtraction - Returns the difference of two operands
*	Multiplication - Returns the product of operands
/	Division – Returns the result of division operation
%	Remainder - Returns the remainder from a division operation

- Following code snippet demonstrates the use of arithmetic operators:

```
...
x = 2 + 3; // Returns 5
y = 8 - 5; // Returns 3
x = 5 * 2; // Returns 10
x = 5/2; // Returns 2
y = 10 % 3; // Returns 1
...
```

Unary Operator

- Unary operators require only one operand.
- They increment/decrement the value of a variable by 1, negate an expression, or invert the value of a boolean variable. Following table lists the unary operators:

Operator	Description
+	Unary plus - Indicates a positive value
-	Unary minus - Negates an expression
++	Increment operator - Increments the value of a variable by 1
--	Decrement operator - Decrements the value of a variable by 1
!	Logical complement operator - Inverts a boolean value

- The prefix version (++variable) will increment the value before evaluating.
- The postfix version (variable++) will first evaluate and then, increment the original value.
- Following code snippet demonstrates the use of unary operators:

```
...
int i = 5;
int j = i++; // i=6, j=5
int k = ++i; //i=6,k=6
i = - i ; //now i is -6
boolean result = false; //result is false
result = !result; //now result is true
…
```

Conditional Operators

- The conditional operators test the relationship between two operands.
- An expression involving conditional operators always evaluates to a boolean value (that is, either `true` or `false`).

- Following code snippet demonstrates use of conditional operators:

```
public class TestConditionalOperators {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int value1 = 10;
      int value2 = 20;
      // Use of conditional operators
      System.out.print("value1 == value2: ");
      System.out.println(value1 == value2);
      System.out.print("value1 != value2: ");
      System.out.println(value1 != value2);
      System.out.print("value1 > value 2: ");
      System.out.println(value1 > value2);
      System.out.print("value1 < value2: ");
      System.out.println(value1 < value2);
      System.out.print("value1 <= value2: ");
      System.out.println(value1 <= value2);
   }
}
```

Logical Operators

- Logical operators (`&&` and `||`) work on two boolean expressions.
- These operators exhibit short-circuit behavior, which means that the second operand is evaluated only if required.
- Following code snippet demonstrates the use of logical operators:

```
public class TestLogicalOperators {
  /**
  * @param args the command line arguments
  */
  public static void main(String[] args) {
    int first = 10;
    int second = 20;

    // Use of logical operator
    System.out.println((first == 30) && (second == 20));
    System.out.println((first == 30) || (second == 20));
  }
}
```

Bitwise Operators

- Bitwise operators work on binary representations of data.
- These operators are used to change individual bits in an operand.
- Following code snippet demonstrates the use of bitwise operators:

```
public class TestBitwiseOperators {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int x = 23;
      int y = 12;
      //23 = 10111 , 12 = 01100
      System.out.print("x & y: ");
      System.out.println(x & y); // Returns 4 , i.e, 4 = 00100
      System.out.print("x | y: ");
      System.out.println(x | y); // Returns 31, i.e 31 = 11111
      System.out.print("x ^ y: ");
      System.out.println(x ^ y); // Returns 27, i.e 31 = 11011
      int a = 43;
      int b = 1;
      System.out.print("a >> b: ");
      System.out.println(a >> b); // returns 21 , i.e, 21 = 0010101
      System.out.print("a << b: ");
      System.out.println(a << b); //returns 86 , i.e, 86 = 1010110
   }
}
```

Ternary Operator

- The ternary operator (`? :`) is a shorthand operator for an if-else statement.
- It makes your code compact and more readable.
- The syntax to use the ternary operator is as follows:

Syntax

```
expression1 ? expression2 : expression3
```

- Following code snippet demonstrates the use of ternary operator:

```
public class VariableScope {
   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      int value1 = 10;
      int value2 = 20;
      int result;
      boolean someCondition = true;
      result = someCondition ? value1 : value2;
      System.out.println(result);
   }
}
```

Operator Precedence 1-2

- Expressions that are written generally consist of several operators.
- The rules of precedence decide the order in which each operator is evaluated in any given expression.
- Following table lists the order of precedence of operators from highest to lowest in which operators are evaluated in Java:

Order	Operator
1.	Parentheses like ( )
2.	Unary Operators like +, -, ++, --, ~, !
3.	Arithmetic and Bitwise Shift operators like *, /, %, +, -, >>, <<
4.	Relational Operators like >, >=, <, <=, ==, !=
5.	Conditional and Bitwise Operators like &, ^, |, &&, ||,
6.	Conditional and Assignment Operators like ?:, =, *=, /=, +=, -=

- Parentheses are used to change the order in which an expression is evaluated.
- Any part of an expression enclosed in parentheses is evaluated first.
- For example, consider the following expression:

```
2*3+4/2 > 3 && 3<5 || 10<9
```

Operator Precedence 2-2

- The evaluation of the expression based on its operators precedence is as follows:

1. • **(2*3+4/2)** > 3 && 3<5 || 10<9
   • First the arithmetic operators are evaluated.
2. • **((2*3)+(4/2))** > 3 && 3<5 || 10<9
   • Division and Multiplication are evaluated before addition and subtraction.
3. • **(6+2)** >3 && 3<5 || 10<9
4. • **(8>3)** && [3<5] || [10<9]
   • Next to be evaluated are the relational operators all of which have the same precedence.
5. • These are therefore evaluated from left to right.
   • **(True && True) ||** False
6. • The last to be evaluated are the logical operators. && takes precedence over ||.
   • True || False
7. • True

Operator Associativity 1-2

- When two operators with the same precedence appear in an expression, the expression is evaluated, according to its associativity.
- For example, in Java the − operator has left-associativity and `x - y - z` is interpreted as `(x - y) - z`, and = has right-associativity and `x = y = z` is interpreted as `x = (y = z)`.
- Following table shows Java operators and their associativity:

Operator	Description	Associativity
(), ++, --	Parentheses, post increment/decrement	Left to right
++, --, +, -, !,~	Pre increment/decrement unary plus, unary minus, logical NOT, and bitwise NOT	unary minus logical NOT, bitwise NOT
Right to left	*, /, %, +, -	Multiplicative and Additive
Left to right	<<, >>	Bitwise shift
Left to right	<, <, >=, <=, ==, !=	Relational and Equality operators
Left to right	&, ^, |	Bitwise AND, XOR, OR
Left to right	&&, ||	Conditional AND, OR
Left to right	?:	Conditional operator (Ternary)

Operator Associativity 2-2

- Consider the following expression:

```
2+10+4-5*(7-1)
```

1

- The '*' has higher precedence than any other operator in the equation.
- However, as `7-1` is enclosed in parenthesis, it is evaluated first.
- `2+10+4-5*6`

2

- Next, '*' is the operator with the highest precedence.
- Since there are no more parentheses, it is evaluated according to the rules.
- `2+10+4-30`

3

- As '+' and '-' have the same precedence, the left associativity works out.
- `12+4-30`

4

- Finally, the expression is evaluated from left to right.
- `6 - 30`
- The result is `-14`.

Type Casting

Type conversion or typecasting refers to changing an entity of one data type into another.

- For instance, values from a more limited set, such as integers, can be stored in a more compact format.
- There are two types of conversion:

implicit

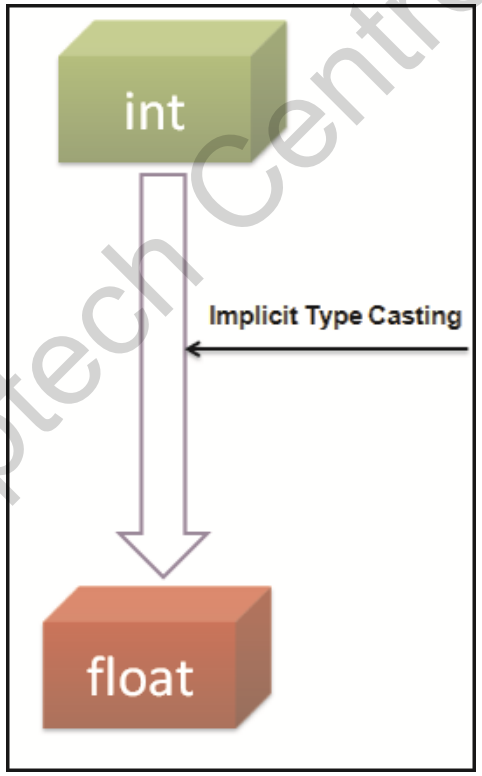
- The term for implicit type conversion is coercion.

explicit

- The most common form of explicit type conversion is known as casting.
- Explicit type conversion can also be achieved with separately defined conversion routines such as an overloaded object constructor.

Implicit Type Casting 1-2

- When a data of a particular type is assigned to a variable of another type, then automatic type conversion takes place.
- It is also referred to as implicit type casting, provided it meets the conditions specified:
  - The two types should be compatible
  - The destination type should be larger than the source
- Following figure shows the implicit type casting:



Implicit Type Casting 2-2

- The primitive numeric data types that can be implicitly cast are as follows:

  - `byte` (8 bits) to `short`, `int`, `long`, `float`, `double`
  - `short` (16 bits) to `int`, `long`, `float`, `double`
  - `int` (32 bits) to `long`, `float`, `double`
  - `long` (64 bits) to `float`, `double`

- This is also known as the type promotion rule.
- The type promotion rules are listed as follows:

  - All `byte` and `short` values are promoted to `int` type.
  - If one operand is `long`, the whole expression is promoted to `long`.
  - If one operand is `float` then, the whole expression is promoted to `float`.
  - If one operand is `double` then, the whole expression is promoted to `double`.

Explicit Casting

- A data type with lower precision, such as `short`, can be converted to a type of higher precision, such as `int`, without using explicit casting.
- However, to convert a higher precision data type to a lower precision data type, such as `float` to `int` data type, an explicit cast is required.
- The syntax for explicit casting is as follows:

Syntax

```
(target data type) value;
```

- Following code snippet adds a `float` value to an `int` and stores the result as an integer:

```
...
float a = 21.3476f;
int b = (int) a + 5;
...
```

- The `float` value in **a** is converted into an integer value **21**.
- It is then, added to **5**, and the resulting value, **26**, is stored in **b**.
- This type of conversion is known as truncation.
- The fractional component is lost when a floating-point is assigned to an integer type, resulting in the loss of precision.

Summary

- Variables store values required in the program and should be declared before they are used. In Java, variables can be declared within a class, method, or within any block.
- Data types determine the type of values that can be stored in a variable and the operations that can be performed on them.
- Data types in Java are divided mainly into primitive types and reference types.
- A literal signifies a value assigned to a variable in the Java program. Java SE 7 supports the use of the underscore characters (_) between the digits of a numeric literal.
- The output of the Java program can be formatted using three ways: print() and println(), printf(), format().
- Operators are symbols that help to manipulate or perform some sort of function on data.
- Parentheses are used to change the order in which an expression is evaluated.
- The type casting feature helps in converting a certain data type to another data type.