School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-3: Object-Oriented Problem Solving (Part-I)

Prerequisite: CSE 1101
Semester: Summer 2024

Object-Oriented Problem Solving

Programming Fundamentals (Part I)

Based on sections from chapters 2, 3 & 4 of
"Introduction to Java Programming" by Y. Daniel Liang.

Outline

- Identifiers (2.4)
- Variables (2.5)
- Assignment statement (2.6)
- Named Constants (2.7)
- Naming Conventions (2.8)
- Numeric Data Types and Operations (2.9)
- Numeric Literals (2.10)
- Evaluating Expressions and Operator Precedence (2.11)
- Augmented Assignment Operators (2.13)
- Increment and Decrement Operator (2.14)
- Numeric Type Conversion (2.15)
- Boolean Data Type (3.2)
- Character Data Type and Operations (4.3)
- The String Type (4.4)

Identifiers

- Identifiers are the names of things that appear in the program.
  - Names of variables, constants, methods, classes, packages…
- All identifiers must obey the following rules:
  - An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar sign ($).
  - Cannot start with a digit.
  - Cannot be a reserved word (e.g., abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, final, finally, float, for, goto, if, implements, import, etc.).
  - Cannot be true, false, or null.
  - Can be of any length.
- Examples of legal identifiers: $2, area, Area, S_3.
- Examples of illegal identifiers: 2A, d+4, S#6.

A simple Java code to show identifiers: java

```
public class IdentifierExample {

    // Class-level identifiers
    private int classVariable = 10;

    // Method with parameter identifiers
    public void printMessage(String message) {
        // Local variable identifiers
        int localVar = 5;

        System.out.println(message);
        System.out.println("Local variable value: " + localVar);
        System.out.println("Class variable value: " + classVariable);
    }

    // Main method with identifiers
    public static void main(String[] args) {
        IdentifierExample example = new IdentifierExample();

        // Method call with argument identifier
        example.printMessage("Hello, identifiers!");
    }
}
```

Output:

```
Hello, identifiers!
Local variable value: 5
Class variable value: 10
```

1. **Class Definition:**

IdentifierExample: This is the name of the class. In Java, class names are identifiers and should start with an uppercase letter by convention.

2. **Class-level Identifier:**
classVariable: This is a private instance variable (field) of type int. It's an identifier used to store and access data within the class.

3. **Method Definition:**
printMessage: This is a method identifier. Method names in Java are also identifiers and should follow camelCase convention.

4. Parameter Identifier:
message: This is a parameter identifier for the printMessage method. Parameters are used to pass values into methods.

5. Local Variable Identifier:
localVar: This is a local variable identifier declared inside the printMessage method. Local variables are defined within methods, constructors, or blocks and have limited scope.

6. Main Method:
main: This is the main method identifier. In Java, main is the entry point of any standalone Java application.

7. Argument Identifier:
"Hello, identifiers!": This is a string literal argument passed to the printMessage method when calling it on the example object.

Variables

- Variables are used to represent values that may be changed in the program.
  - They are used to store values to be used later in the program.
- To use a variable, you declare it by telling the compiler its name as well as what type of data it can store.
- The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type.
  - The syntax for declaring a variable:

  datatype variableName
- Examples of variable declarations:
  - *int count;*
  - *double rate;*
  - *char letter;*
  - *boolean found;*

Variables (Cont.)

- Several variables can be declared together:
  - *int count, limit, numberOfStudents;*
- When a variable is declared, the compiler allocates memory space for the variable based on its data type.

A Java code snippet that declares and initializes variables of different types: int, double, char, and boolean.

```
public class VariableExample {

    public static void main(String[] args) {
        // Declaring and initializing variables
        int count = 5;
        double rate = 3.5;
        char letter = 'A';
        boolean found = true;

        // Printing the values of variables
        System.out.println("Count: " + count);
        System.out.println("Rate: " + rate);
        System.out.println("Letter: " + letter);
        System.out.println("Found: " + found);

        // Modifying the values of variables
        count = 10;
        rate = 2.75;
        letter = 'B';
        found = false;

        // Printing modified values
        System.out.println("Updated Count: " + count);
        System.out.println("Updated Rate: " + rate);
        System.out.println("Updated Letter: " + letter);
        System.out.println("Updated Found: " + found);
    }
}
```

Count: 5
Rate: 3.5
Letter: A
Found: true
Updated Count: 10
Updated Rate: 2.75
Updated Letter: B
Updated Found: false

Assignment Statement

- An assignment statement designates a value for a variable.
- The *equal sign (=)* is used as the assignment operator.
- Examples:

```
x = 1;
x = x+1;
area = radius * radius * 3.14159;
```

Assignment Statement (Cont.)

- Variables can be declared and initialized in one step:

  int count = 0;

  char letter = 'a';

  boolean found = false;

  int i = 1, j = 2;

- *int count = 0;* is equivalent to the following two statements:

  int count;

  count = 0;

Assignment Statement (Cont.)

- An assignment statement can be used as an expression in Java:

  System.out.println(x=1);

- A value can be assigned to multiple variables:

  i = j = k = 1;

- In an assignment statement the data type of the variable on the left must be compatible with the data type of the value on the right.

  Except if *type casting* is used.

Named Constants

- A *named constant* is an identifier that represents a permanent value.
- A *constant* must be declared and initialized in the same statement.
- The syntax for declaring a constant:

  final datatype CONSTANT_NAME = value;

- Example:

  final double PI = 3.14159;

Named Constants (Cont.)

- There are three benefits of using constants:

(1) You don't have to repeatedly type the same value if it is used multiple times.

(2) If you have to change the constant value (e.g., from **3.14** to **3.14159** for **PI**), you need to change it only in a single location in the source code; and

(3) A descriptive name for a constant makes the program easy to read.

```
public class ConstantsExample {

    // Named constants declaration
    public static final int MAX_VALUE = 100;
    public static final double PI = 3.14159;
    public static final String GREETING = "Hello, World!";
    public static final boolean DEBUG_MODE = true;

    public static void main(String[] args) {
        // Using named constants
        System.out.println("Max Value: " + MAX_VALUE);
        System.out.println("PI Value: " + PI);
        System.out.println("Greeting: " + GREETING);
        System.out.println("Debug Mode: " + DEBUG_MODE);

        // Attempting to modify a constant (will cause compilation error)
        // MAX_VALUE = 200; // Uncommenting this line will result in a compilation
error
    }
}
```

Max Value: 100
PI Value: 3.14159
Greeting: Hello, World!
Debug Mode: true

1. Named Constants Declaration:
public static final int MAX_VALUE = 100;
: Declares a constant named MAX_VALUE of type int with a value of 100. The final keyword ensures that this value cannot be changed once initialized.

public static final double PI = 3.14159;
: Declares a constant named PI of type double with a value approximating π (pi). Again, final ensures it remains constant.

public static final String GREETING = "Hello, World!";
: Declares a constant named GREETING of type String with the value "Hello, World!".

public static final boolean DEBUG_MODE = true;
: Declares a constant named DEBUG_MODE of type boolean with the value true.

2. Using Named Constants:
Within the main method, each constant is printed using System.out.println() to demonstrate their values.

3. Modifying a Constant:
An attempt to modify a constant (MAX_VALUE = 200;) is shown but commented out. Modifying a constant declared with final would result in a compilation error.

Naming Conventions

- Sticking with the Java naming conventions makes your programs easy to read and avoids errors.
- Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program.
- Use lowercase for variables and methods.
  - E.g. radius, count.
  - If a name consists of several words, concatenate them, make the first word lowercase and capitalize the first letter of each subsequent word.
    - E.g. numberOfStudents.
- Capitalize the first letter of each word in a class name.
  - E.g. ComputeAread, String.
- Capitalize every letter in a constant, and use underscores between words.
  - PI, MAX_VALUE.

Numeric Data Types

- Java has six built-in numeric data types.

	Name	*Range*	*Storage Size*
Integers	`byte`	-2^7 to 2^7-1 (−128 to 127)	8-bit signed
	`short`	-2^{15} to $2^{15}-1$ (−32768 to 32767)	16-bit signed
	`int`	-2^{31} to $2^{31}-1$ (−2147483648 to 2147483647)	32-bit signed
	`long`	-2^{63} to $2^{63}-1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed
Floating Point Numbers	`float`	Negative range: −3.4028235E+38 to −1.4E−45 Positive range: 1.4E−45 to 3.4028235E+38	32-bit IEEE 754
	`double`	Negative range: −1.7976931348623157E+308 to −4.9E−324 Positive range: 4.9E−324 to 1.7976931348623157E+308	64-bit IEEE 754

Numeric Operators

Name	*Meaning*	*Example*	*Result*
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

Numeric Literals: Integrals

- A literal is a constant value that appears directly in a program.
  - int numberOfYears = *34*;
  - double weight = *0.305*;
- An integer literal can be assigned to an integer variable as long as it can fit into the variable.
  - Otherwise a compile error occurs.
  - e.g. *byte b = 128;* will cause a compilation error.
- To denote an integer literal of the long type, append letter L or l to it.
  - e.g. *2147483648L*

Numeric Literals: Floating Points

- Floating point literals are written with a decimal point.
- By default, a floating point literal is treated as a double type value.

  5.0 is considered a double value.
- You can make a number a float by appending the letter f or F.

  e.g. *100.2F*

Numeric Literals: Floating Points (Cont.)

- Double type values are more accurate than the float type values.

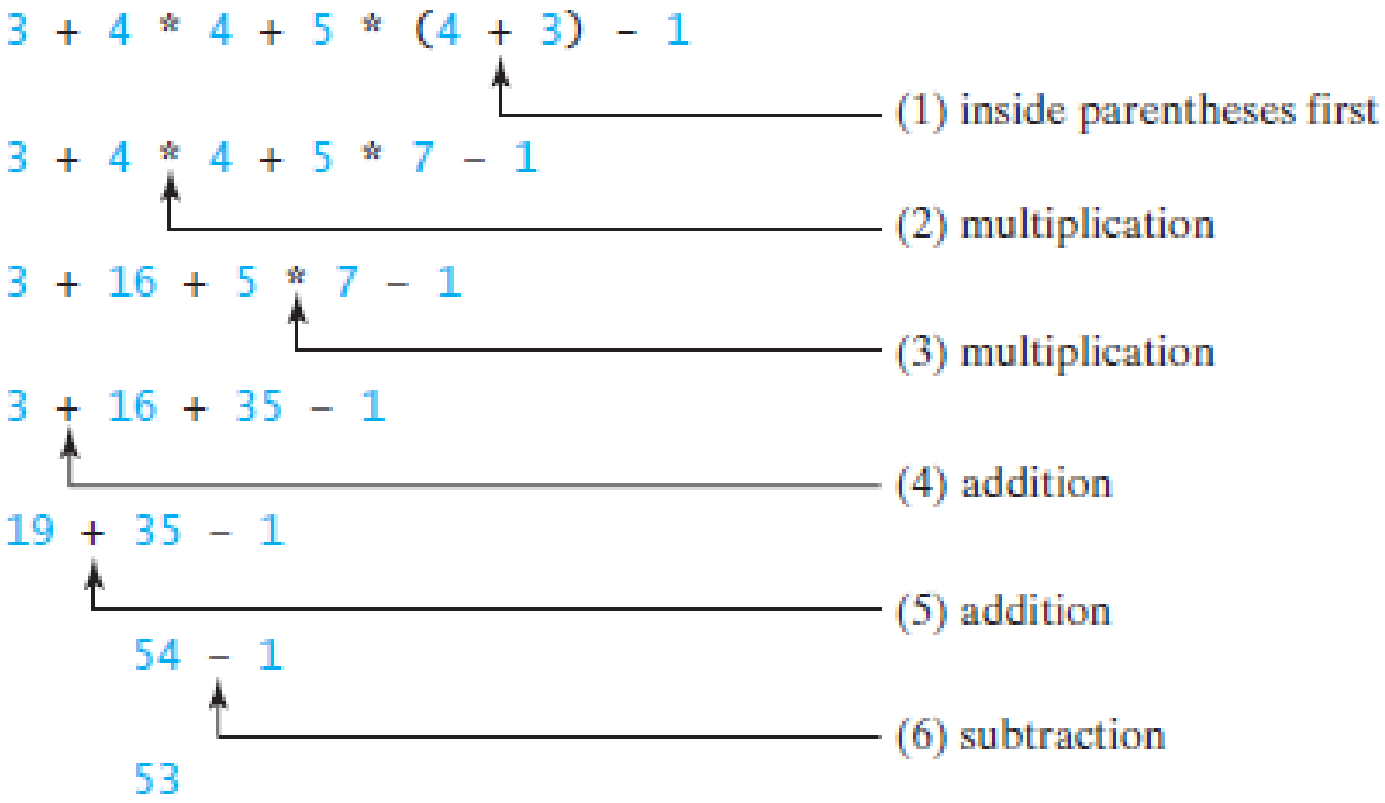
  System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);

  Displays *1.0 / 3.0 is 0.3333333333333333*

  System.out.println("1.0 / 3.0 is " + 1.0F / 3.0F);

  Displays *1.0 / 3.0 is 0.33333334*

Evaluating Expressions and Operator Precedence



Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and Decrement Operators

Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

The pre-increment operator int j = ++i;
Increases the value of i by 1 before the value is used in any expression. This means that if i was 1 before the operation, it becomes 2 after the operation, and the new value (2) is used in the assignment.

```
public class PreIncrementExample {
    public static void main(String[] args) {
        int i = 1;       // Step 1: Initialize i to 1
        int j = ++i;     // Step 2: Pre-increment i and assign the result to j

        // Output the values of i and j
        System.out.println("Value of i: " + i); // i is now 2
        System.out.println("Value of j: " + j); // j is also 2
    }
}
```

```
Value of i: 2
Value of j: 2
```

Post-increment Operation: int j = i++;

The post-increment operator (i++) increments the value of i by 1, but the original value of i is used in the assignment before the increment. Before i++, i is 1.The value of i (which is 1) is assigned to j. After the assignment, i is incremented to 2.

```
public class PostIncrementExample {
    public static void main(String[] args) {
        int i = 1;       // Step 1: Initialize i to 1
        int j = i++;     // Step 2: Assign the value of i to j, then increment i

        // Output the values of i and j
        System.out.println("Value of i: " + i); // i is now 2
        System.out.println("Value of j: " + j); // j is 1
    }
}
```

```
Value of i: 2
Value of j: 1
```

Pre-decrement Operation: int j = --i;

The pre-decrement operator (--i) decrements the value of i by 1 before the value is used in the assignment. The decremented value of i is then assigned to j. Before --i, i is 1.The --i operation decrements i to 0.The new value of i (which is 0) is then assigned to j.

```
public class PreDecrementExample {
    public static void main(String[] args) {
        int i = 1;       // Step 1: Initialize i to 1
        int j = --i;     // Step 2: Pre-decrement i and assign the result to j

        // Output the values of i and j
        System.out.println("Value of i: " + i); // i is now 0
        System.out.println("Value of j: " + j); // j is also 0
    }
}
```

Value of i: 0
Value of j: 0

Post-decrement Operation: int j = i--;

The post-decrement operator (i--) decrements the value of i by 1, but the original value of i is used in the assignment before the decrement. Before i--, i is 1.The value of i (which is 1) is assigned to j. After the assignment, i is decremented to 0.

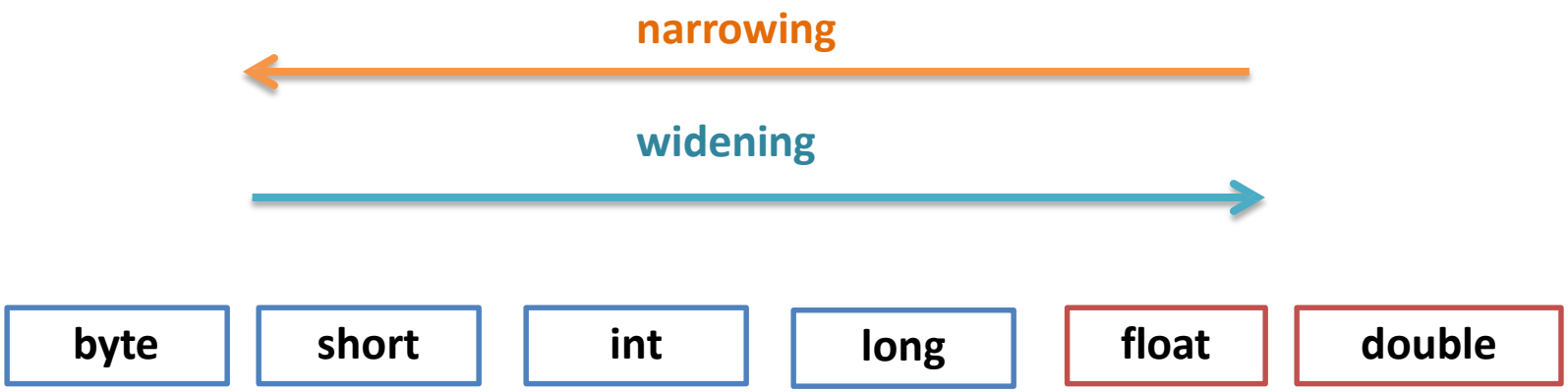
```
public class PostDecrementExample {
    public static void main(String[] args) {
        int i = 1;       // Step 1: Initialize i to 1
        int j = i--;     // Step 2: Assign the value of i to j, then decrement i

        // Output the values of i and j
        System.out.println("Value of i: " + i); // i is now 0
        System.out.println("Value of j: " + j); // j is 1
    }
}
```

Value of i: 0
Value of j: 1

Numeric Type Conversions

- You can always assign a value to a numeric variable whose type supports a larger range of values.
  - You can assign a **long** value to a **float** variable.
- You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*.
- *Casting* is an operation that converts a value of one data type into a value of another data type.
  - *Widening a type* is casting a type with a small range to a type with a larger range.
    - e.g. Integer to floating point: *3 * 4.5* is same as *3.0 * 4.5*.
  - *Narrowing a type* is casting a type with a large range to a type with a smaller range.
    - e.g. floating point to integer:
      System.out.println ((int)1.7);
- Java automatically widens a type, but you must narrow a type explicitly.

Numeric Type Conversions



boolean Data Type

- A *boolean* data type declares a variable with the value *true* or *false*.
- Boolean expressions represent conditions that are used to make decisions in the program.

Comparison Operators

Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	less than	`radius < 0`	`false`
`<=`	≤	less than or equal to	`radius <= 0`	`false`
`>`	>	greater than	`radius > 0`	`true`
`>=`	≥	greater than or equal to	`radius >= 0`	`true`
`==`	=	equal to	`radius == 0`	`false`
`!=`	≠	not equal to	`radius != 0`	`true`

- The result of a comparison is a boolean value: *true* or *false*.
- Note that the equality comparison operator is two equal signs (==), not a single equal sign (=); this is the assignment operator

The Character Data Type

- The *character* data type represents a single character.
- A character literal is enclosed in single quotation marks.
- Examples:
  - *char letter = 'A';*
  - *char numChar = '4';*
- Java uses *Unicode* which is designed as a 16-bit character encoding.

Unicode for Commonly Used Characters

Characters	*Code Value in Decimal*	*Unicode Value*
`'0'` to `'9'`	48 to 57	\u0030 to \u0039
`'A'` to `'Z'`	65 to 90	\u0041 to \u005A
`'a'` to `'z'`	97 to 122	\u0061 to \u007A

Character Literals

Escape Character	*Name*
\b	Backspace
\t	Tab
\n	Linefeed
\f	Formfeed
\r	Carriage Return
\\	Backslash
\"	Double Quote

Casting between *char* and Numeric Types

- A *char* can be cast into any numeric type, and vice versa.
- When an integer is cast into a **char**, only its lower 16 bits of data are used; the other part is ignored.
  - *char ch = (char)0XAB0041;*
  - *System.out.println(ch); // ch is character A*
- When a floating-point value is cast into a *char*, the floating-point value is first cast into an *int*, which is then cast into a *char*.
  - *char ch = (char)65.25;*
  - *System.out.println(ch); // ch is character A*

In Java, the expression char ch = (char) 0xAB0041; involves casting an integer literal 0xAB0041 to a char type.

```
public class CharExample {
    public static void main(String[] args) {
        char ch = (char) 0xAB0041;

        // Output the value of ch
        System.out.println("Value of ch: " + ch); // This will print the Unicode character corresponding to 0xAB0041
    }
}
```

Value of ch: A

In Java, the expression char ch = (char) 65.25; involves casting a floating-point number (double) 65.25 to a char type. Let's break down what happens in this context:

```
public class CharExample {
    public static void main(String[] args) {
        char ch = (char) 65.25;
        System.out.println("Character: " + ch); // Output:
Character: A
    }
}
```

Character: A

Casting between *char* and Numeric Types (Cont.)

- When a *char* is cast into a numeric type, the character's Unicode is cast into the specified numeric type.
  - *int i = (int)'A';*
  - *System.out.println(i); // i is 65*
- Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used.
  - *byte b = 'a';*
  - *int i = 'a';*
- Any positive integer between **0** and **FFFF** in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a *char* explicitly.

In Java, the expression int i = (int) 'A'; involves casting a char type ('A') to an int type. Let's break down what happens in this context:

```
public class IntExample {
    public static void main(String[] args) {
        int i = (int) 'A';
        System.out.println("Integer value of 'A': " + i); // Output:
Integer value of 'A': 65
    }
}
```

Integer value of 'A': 65

The String Type

- A string is a sequence of characters.
- To represent a string of characters, use the data type called *String*:
  - e.g. *String message = "Welcome to Java";*
- *String* is a predefined class in the Java library.
- The String type is not a *primitive* type. It is known as a *reference* type.
- A string literal must be enclosed on quotation marks (" ").

The String Type (Cont.)

- The *plus sign* (*+*) is the *concatenation* operator if at least one of the operands is a string.
  - If one of the operands is a non string (e.g. a number), the non string value is converted into a string and concatenated with the string.
  - Examples:
    - *String message = "Welcome " + "to " + "Java!";*
      message becomes: Welcome to Java!
    - *String s = "Chapter" + 2;*
      s becomes: Chapter2
    - *String appendix = "Appendix" + 'B';*
      appendix becomes: AppendixB
- If neither of the operands is a string, the *plus sign* (*+*) is the *addition* operator.

In Java, the expression String message = "Welcome" + "to" + "Java!"; involves string concatenation using the + operator with string literals.

```
public class StringConcatenationExample {
    public static void main(String[] args) {
        String message = "Welcome" + "to" + "Java!";
        System.out.println("Message: " + message); // Output:
Message: Welcome to Java!
    }
}
```

Message: Welcome to Java!

In Java, the expression String message = "Lecture" + 3; involves concatenating a string literal "Lecture" with an integer 3 using the + operator.

```
public class StringConcatenationExample {
    public static void main(String[] args) {
        String message = "Lecture" + 3;
        System.out.println("Message: " + message); // Output: Message: Lecture3
    }
}
```

Message: Lecture3