



Week 3 Java Basics (2/2)

4009 DATA STRUCTURES & ALGORITHM (DSA)

JAVA BASICS PART 2 OF 2

Learning Outcomes

- ❖ Identify statement block scope and use of local variables of Java programs
- ❖ Differentiate between the different control flows such if statements, switch statement and loops
- ❖ Discuss what are literals and operators
- ❖ Understand the syntax and expected simple input and output of a Java Program.
- ❖ Discuss the concept of casting variables
- ❖ Implement a single Java program with methods, control flows, string concatenation and expressions.

Content Overview

- ❖ Recap the Main Method, statement blocks and local variables
- ❖ Expressions
 - Literals
 - Operators
- ❖ Casting
- ❖ Control flow
 - If statements
 - Switch statement
 - Loops
- ❖ Simple input & Output (IO)

The Main Method

The primary control for an application in Java must begin in some class with the execution of a special method named main. This method must be declared as follows:

```
public static void main(String[ ] args) {  
    // main method body...  
}
```

The args parameter is an array of String objects, that is, a collection of indexed strings, with the first string being args[0], the second being args[1], and so on.

```
// Create a Main class  
public class Main {  
    int x;  
  
    // Create a class constructor for the Main class  
    public Main() {  
        x = 5;  
    }  
  
    public static void main(String[] args) {  
        Main myObj = new Main();  
        System.out.println(myObj.x);  
    }  
}
```

5

Statement Blocks & Local Variables


The body of a method is a *statement block* which is a sequence of statements and declarations to be performed between the **braces** “{“ and “}”.

Method bodies and other statement blocks can themselves have statement blocks nested inside of them.

In addition to statements that perform some action like calling the method of some object, statement blocks can contain declarations of *local variables*.

There are two ways of declaring local variables:

1. *type name;*
2. *type name = initial_value;*



```
// Create a Main class
public class Main {
    int x;

    // Create a class constructor for the Main class
    public Main() {
        x = 5;
    }

    public static void main(String[] args) {
        Main myObj = new Main();
        System.out.println(myObj.x);
    }
}
```

5

Statement Blocks & Local Variables

Every Java program has a **class name** which **must match** the **filename** of the program. *Example `public class Main{} is the class, therefore, the program file is called Main.java.`*

The **curly braces {}** marks the **beginning and the end of a block of code**.

System is a **built-in Java class** that contains useful members such as `out`, which is short for “**output**”.

The **`println()` method** is short for “**print line**” used to print a value to the screen (or a file).

```
// Create a Main class
public class Main {
    int x;

    // Create a class constructor for the Main class
    public Main() {
        x = 5;
    }

    public static void main(String[] args) {
        Main myObj = new Main();
        System.out.println(myObj.x);
    }
}
```

5

Expressions

Variables and Constants are used in expressions to define new values and to modify variables.

Expressions involve the use of **literals, variables and operators**.

Literals

A literal is any “constant” value that can be used in an assignment or other expression. Java allows the following kinds of literals:

- The **null** object reference
- **Boolean:** true or false
- **Integer:** The default for integer like 176 or – 52 is that it is type **int** which is a 32-bit integer/size
- **Long integer:** literal must end with an L, example: 176L or 176l which is 64-bit integer/size
- **Floating point:** the default floating-numbers like 3.1415F or 3.1415f which is 32-bit size
- **Double:** used for more decimal numbers like 3.1415 which is 64-bit size
- **Character:** assumed to be taken by Unicode alphabet at 16-bit size
- **String literal:** a sequence of characters enclosed in double quotes. For example: “dogs cannot climb trees”

Literals

Special character constants

'\n'	New line
'\t'	Tab
'\b'	Backspace
'\r'	Return
'\f'	Form feed
'\\'	Backslash
'\''	Single quote
'\"'	Double quote

Operators

Java expressions involve composing literals and variables with operators.

The Assignment Operator =

Used to assign a value to an instance variable or local variable.

Variable = expression

Arithmetic Operator

Binary arithmetic operators in Java

Operator	Operator name	Use	Description
+	Addition	op1 + op2	Adds op1 and op2
-	Subtraction	op1 - op2	Subtracts op2 from op1
*	Multiplication	op1 * op2	Multiplies op1 by op2
/	Division	op1 / op2	Divides op1 by op2
%	Modulus operator (remainder)	op1 % op2	Computes the remainder of dividing op1 by op2

Increment & Decrement Operators

Increment and decrement operators used into Java.

Example: If an operator (++) here is used in front of a variable reference then 1 is added to the variable and its value is read into the expression.

Operator	Use	Description
+	+ op	Promotes op to int if it's a byte, short, or char
-	- op	Arithmetically negates op
++	op ++	Increments op by 1; evaluates to the value of op before it was incremented
++	++ op	Increments ... (after)
--	op --	Decrements ... (before)
--	-- op	Decrements ... (after)

```
Int x = 8;  
Int y = x++;  
Int z = 9 + x++;
```

Comparison Operators

Used in logical statements to determine equality or difference between variables or values.

Operator	Operator name	Use	Returns <i>true</i> if
>	Greater than	op1 > op2	op1 is greater than op2
>=	Greater than or equal to	op1 >= op2	op1 is greater than or equal to op2
<	Less than	op1 < op2	op1 is less than op2
<=	Less than or equal to	op1 <= op2	op1 is less than or equal to op2
==	Equal to	op1 == op2	op1 and op2 are equal
!=	Not equal to	op1 != op2	op1 and op2 are not equal

Logical Operators

Used to determine the logic between variables or values

Operator	Use	Returns <i>true</i> if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
	op1 op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false

String Concatenation

Concatenate is defined as an operation to join or link something together.

String processing involves dealing with Strings (- an object that represents a number of characters values). Therefore, the primary operating for combining strings is called Concatenation.

For example:

A string P and a string Q combines them into a new string called R

P = “kilo” , Q = “meters” , R =??

R = P + Q

R = “kilo” + “meters”;

R = “kilometers”;

Casting and Autoboxing/Unboxing in expressions

Casting is an operation that allows us to change the type of a value.

In essence, we can take a value of one type and **cast** it into an equivalent value of another type.

There are two forms of casting in Java: *explicit casting and implicit casting*.

Explicit casting

Java supports an explicit casting syntax with the following form:

(type) exp

where type is the type that we would like the expression exp to have. This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type.

Implicit casting

implicit cast based upon the context of an expression. For example, you can perform a widening cast between primitive types (such as from an int to a double), without explicit use of the casting operator

Example:

```
int i = 42;  
double d = i;    // i gets value 42.0
```

Casting and Autoboxing/Unboxing in expressions

Explicit casting example.

```
double d1 = 3.2;  
double d2 = 3.9999;  
int i1 = (int) d1;    //i1 gets value 3  
int i2 = (int) d2;    //i2 gets value 3  
Double d3 = (double) i2; //d3 gets value  
3.0
```

Implicit casting example.

```
int i1 = 42;  
double d1 = i1; // i1 gets value 42.0
```


Control Flow

Control flow in Java is similar to that of other high-level languages.

We review the basic structure and syntax of control flow in Java in this section, including **method returns**, **if statements**, **switch statements**, **loops**, and **restricted forms of “jumps”** (the **break** and **continue statements**).

The if statement

The If Statement

The syntax of a simple **if** statement is as follows:

```
if (booleanExpression)  
    trueBody  
else  
    falseBody
```

where *booleanExpression* is a boolean expression and *trueBody* and *falseBody* are each either a single statement or a block of statements enclosed in braces (“{” and “}”).

```
if (firstBooleanExpression)  
    firstBody  
else if (secondBooleanExpression)  
    secondBody  
else  
    thirdBody
```

If the first boolean expression is false, the second boolean expression will be tested, and so on. An if statement can have an arbitrary number of else if parts. Braces can be used for any or all statement bodies to define their extent.

Switch statements

Java provides for multiple-value control flow using the switch statement, which is especially useful with enum types.

```
switch (d) {  
    case MON:  
        System.out.println("This is tough.");  
        break;  
    case TUE:  
        System.out.println("This is getting better.");  
        break;  
    case WED:  
        System.out.println("Half way there.");  
        break;  
    case THU:  
        System.out.println("I can see the light.");  
        break;  
    case FRI:  
        System.out.println("Now we are talking.");  
        break;  
    default:  
        System.out.println("Day off!");  
}
```

The **switch** statement evaluates an integer, string, or enum expression and causes control flow to jump to the code location labeled with the value of this expression.

If there is no matching label, then control flow jumps to the location labeled “**default.**” This is the only explicit jump performed by the **switch** statement, however, so flow of control “falls through” to the next case if the code for a case is not ended with a **break** statement (which causes control flow to jump to the end).

Loops

Another important control flow mechanism in a programming language is looping.

Java provides for three types of loops

1. While loops
2. For loops
3. Do-While loops

While Loops

The simplest kind of loop in Java is a while loop

while (*booleanExpression*)
 loopBody

The execution of a while loop begins with a test of the boolean condition. If that condition evaluates to true, the body of the loop is performed. After each execution of the body, the loop condition is retested and if it evaluates to true, another iteration of the body is performed. If the condition evaluates to false when tested (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop

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

Do-While loop

Java has another form of the while loop that allows the boolean condition to be checked at the end of each pass of the loop rather than before each pass. This form is known as a do-while loop, and has syntax shown below:

do
 loopBody
while (*booleanExpression*)

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

A consequence of the do-while loop is that its body always executes at least once. (In contrast, a while loop will execute zero times if the initial condition fails.) This form is most useful for a situation in which the condition is ill-defined until after at least one pass

For Loop

Another kind of loop is the **for loop**.

The for-loop syntax consists of four sections—an initialization, a boolean condition, an increment statement, and the body—although any of those can be empty.

```
for (int counter =0; condition; increment){  
    loops statement  
}
```

```
{  
    initialization;  
    while (booleanCondition) {  
        loopBody;  
        increment;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        for (int i = 0; i <= 10; i = i + 2) {  
            System.out.println(i);  
        }  
    }  
}
```

Explicit Control-Flow Statements

Java also provides statements that cause explicit change in the flow of control of a program.

Return from a method

If a Java method is declared with a return type of **void**, then flow of control returns when it reaches the last line of code in the method or when it encounters a **return** statement (with no argument).

If a method is declared with a return type, however, the method must exit by returning an appropriate value as an argument to a **return** statement. It follows that the **return** statement *must* be the last statement executed in a method, as the rest of the code will never be reached.

*Note that there is a significant difference between a statement being the last line of code that is **executed** in a method and the last line of code in the method itself*

Explicit Control-Flow Statements

The following (correct) example illustrates returning from a method:

```
public double abs (double value) {  
    if (value < 0)                // value is negative,  
        return -value;           // so return its negation  
    return value;                 // return the original nonnegative value  
}
```

In the example above, the line **return** -value; is clearly not the last line of code that is written in the method, but it may be the last line that is executed (if the original value is negative). Such a statement explicitly interrupts the flow of control in the method. There are two other such explicit control-flow statements, which are used in conjunction with loops and switch statements.

Explicit Control-Flow Statements

The break Statement

Can be used to “break” out of the innermost **switch**, **for**, **while**, or **do-while** statement body.

When it is executed, a break statement causes the flow of control to jump to the next line after the loop or **switch** to the body containing the **break**.

```
public class Main {  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; i++) {  
            if (i == 4) {  
                break;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

0
1
2
3

The continue Statement

A continue statement can be used within a loop.

It causes the execution to skip over the remaining steps of the current iteration of the loop body, but then, unlike the break statement, the flow of control returns to the top of the loop, assuming its condition remains satisfied.

```
public class Main {  
    public static void main(String[] args) {  
        for (int i = 0; i < 10; i++) {  
            if (i == 4) {  
                continue;  
            }  
            System.out.println(i);  
        }  
    }  
}
```

0
1
2
3
5
6
7
8
9

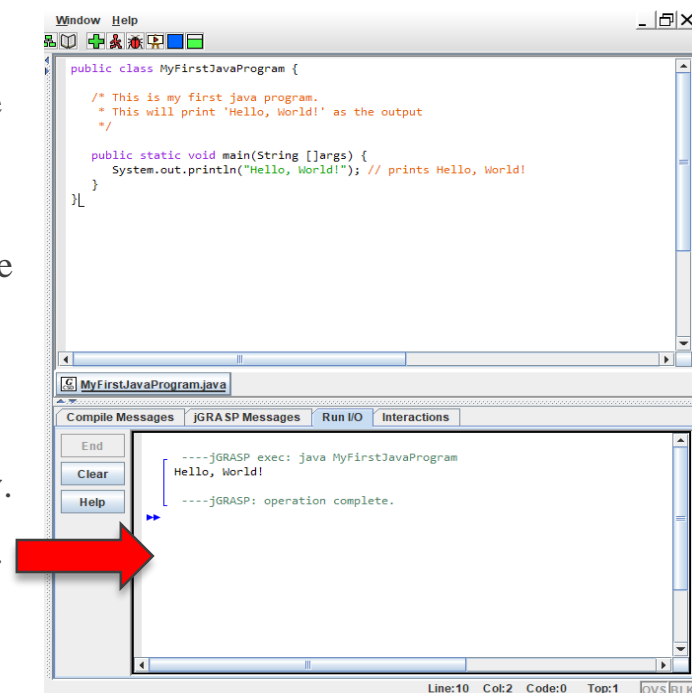
Simple Input & Output

Java provides a rich set of classes and methods for performing input and output within a program.

There are classes in Java for doing graphical user interface design, complete with pop-up windows and pull-down menus, as well as methods for the display and input of text and numbers. Java also provides methods for dealing with graphical objects, images, sounds, Web pages, and mouse events (such as clicks, mouse overs, and dragging). Moreover, many of these input and output methods can be used in either stand-alone programs or in applets.

How simple input and output can be done in Java in this section.

Simple input and output in Java occurs within the Java console window. Depending on the Java environment we are using, this window is either a special pop-up window that can be used for displaying and inputting text, or a window used to issue commands to the operating system (such windows are referred to as shell windows, command windows, or terminal windows).



Simple Output Methods

Java provides a built-in static object, called **System.out**, that performs output to the “standard output” device. Most operating system shells allow users to redirect standard output to files or even as input to other programs, but the default output is to the Java console window.

The `System.out` object is an instance of the `java.io.PrintStream` class. This class defines methods for a buffered output stream, meaning that characters are put in a temporary location, called a *buffer*, which is then emptied when the console window is ready to print characters.

The `java.io.PrintStream` class provides the following methods for performing simple output:

`print(String s)`: Print the string `s`.

`print(Object o)`: Print the object `o` using its `toString` method.

`print(baseType b)`: Print the base type value `b`.

`println(String s)`: Print the string `s`, followed by the newline character.

`println(Object o)`: Similar to `print(o)`, followed by the newline character.

`println(baseType b)`: Similar to `print(b)`, followed by the newline character.

An Output Example

Consider, for example, the following code fragment:

```
System.out.print("Java values: ");
System.out.print(3.1416);
System.out.print(' ', ' ');
System.out.print(15);
System.out.println(" (double,char,int).");
```

When executed, this fragment will output the following in the Java console window:

```
Java values: 3.1416,15 (double,char,int).
```

Simple Input using the java.util.Scanner class

Java has a special object, called **System.in**, for performing input from the Java console window.

Technically, the input is actually coming from the “standard input” device, which by default is the computer keyboard echoing its characters in the Java console. The System.in object is an object associated with the standard input device.

A simple way of reading input with this object is to use it to create a *Scanner object*, using the expression *new Scanner(System.in)*

The Scanner class has a number of convenient methods that read from the given input stream, one of which is demonstrated in the following program:

Example of simple Input & output program

```
import java.util.Scanner;           // loads Scanner definition for our use

public class InputExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your age in years: ");
        double age = input.nextDouble();
        System.out.print("Enter your maximum heart rate: ");
        double rate = input.nextDouble();
        double fb = (rate - age) * 0.65;
        System.out.println("Your ideal fat-burning heart rate is " + fb);
    }
}
```

When executed, this program could produce the following on the Java console:

```
Enter your age in years: 21
Enter your maximum heart rate: 220
Your ideal fat-burning heart rate is 129.35
```

Topic next week

Week 4 Object Oriented Design

DSA Week 3 activities

Refer to print out materials.

- This week, you are required to complete the questions and two labs.
 - Refer to the print out, answer all week 3 questions.
 - Refer to the print outs, complete the two labs activities using the lab computers.

Note: You can complete the activities in any order, however, make afford to complete and understand everything which prepares you for well for test 1, test 2, Major Assignment, Mid Semester Exam & Final Exam.