Fundamentals of Programming 2

Lecture 2

Aurelia Power, TU Dublin – Blanchardstown Campus, 2019

* Notes based on the Java Oracle Tutorials, Deitel & Deitel, and Horstman

Expressions in Java

- Like in Python, the combination of variables, literal values, operators, and/or method calls is called an **expression**.
- For instance, the right-hand side of an assignment statement can be itself an expression and contain variable names.

```
double price, tax, cost;
price = 500;
tax = 17.5;
cost = price * (1 + tax/100) ;
```

Cost is assigned the value of 587.5

Value of price is 500

Value of tax is 17.5

REMEMBER:

1. first the expression on the right hand side is evaluated: price * (1 + tax/100)
2. then the result is assigned to the variable on the left hand side of the expression cost is assigned 587.5

Expressions in Java

- Like in Python, you can even use the name of a variable on both sides of the assignment expression:

1. On the right: the variable holds the old value (assigned before).
2. On the left : the variable will hold the new value (the result of evaluating the expression on the right).

```
price = price * (1 + tax/100) ;
```

The new value of 'price'

the old value of 'price'

Displaying Text with **printf**

- To display data in a formatted way we can use the **printf** method;
- The printf method allows you more flexibility in how you output your data, for instance, when you want to specify how many digits after the decimal point:

```
System.out.printf("%.2f", 235.99344); // will output 235.99
```

The first argument of the method printf is a format specifier.

The second argument, is the value that needs to be formatted.

- A **format specifier** is a placeholder for a value and specifies the type of data to output.
- Format specifiers begin with a **percent sign (%)** and typically are followed by a character that represents the data type.

Displaying Text with **printf**

- You can have more that one format specifier, and, in that case, you should specify the same number of values/variables to be formatted → you will need **multiple method arguments** that are separated by comma:

System.out.printf("%.3f kg of meat costs %.2f", 2.5444444, 16.3333333333);

% signals the beginning of a format specifier.

3 specifies how many digits after the decimal point

f stands for floating point number so it will output double and float values

Any literal characters that are not format specifiers are printed verbatim

Another format specifier for another floating point number

The 2 values (arguments) which correspond to the format specifiers: 1st argument to the 1st format specifier, 2nd argumnent to the 2nd specifier.

So, the number of arguments the method printf can take may vary.

- // 2.544 kg of meat costs 16.33

Displaying Text with **printf**

- Other format specifiers:
  - **Formatting Strings: %s**

System.out.printf("%s", "We are Year 1"); // will output **We are Year 1**

  - **Formatting Integers: %d**

System.out.printf("%d", 25); // will output **25**

  - **Formatting Characters: %c**

System.out.printf("I am the 7th character in the alphabet %c", 'g'); // will output **25**

  - **Formatting New Line: %n**

System.out.printf("%d%nThe cursor was moved to the next line", 25);

25

The cursor was moved to the next line

- Specifiers can be used to output values of variables:

String str = "Age"; int age = 34;

System.out.printf("%s: %d", str, age);

// will output **Age: 34**

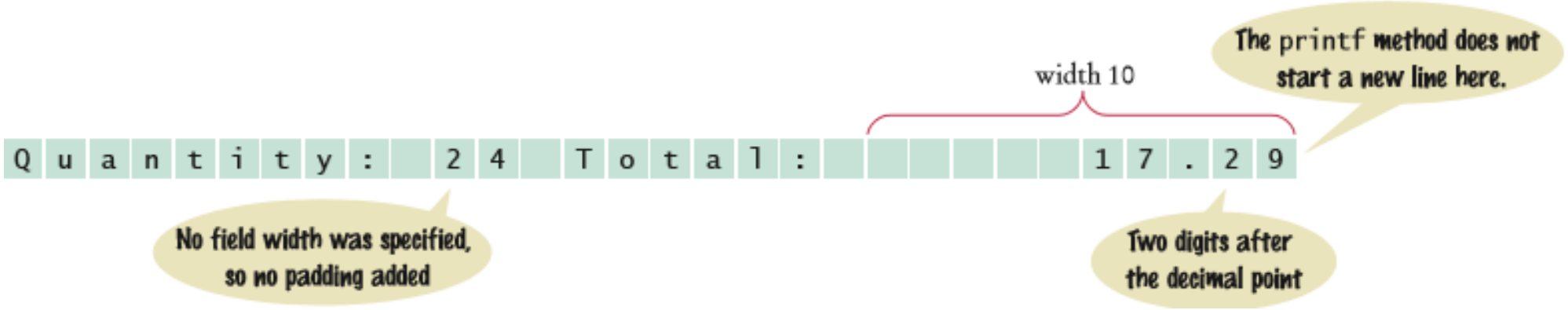
Displaying Text with **printf**

- **NOTE:** If you mix up the values, you will get an error when you run the program:

```
String str = "Age"; int age = 34;

System.out.printf("%s: %d", age, str);

/* will output something like this: Exception in thread "main"
    java.util.IllegalFormatConversionException: d != java.lang.String */
```

- You can specify the number of spaces to allocate for a value just after the **%** sign.

```
int quantity = 24; double total = 17.29;

System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```



Escape Sequences

- Sometimes you need to print out special characters such as double quotation marks, but they are used to enclose String literals.

- To print such characters we can use **escape sequences.**
- Escape sequences are created using the **backslash** (\) which is called an escape character;
- For instance: if we want to print **She said "I will meet you there" on Wednesday**

```
System.out.println("She said \"I will meet you there\" on Wednesday ");
```

- **NOTE**: If you forget to provide the \, your code will not compile:

```
System.out.println("She said "I will meet you there\" on Wednesday "); // not compiling
System.out.println("She said  \"I will meet you there" on Wednesday "); // not compiling
System.out.println("She said  "I will meet you there " on Wednesday "); // not compiling
```

Other Common Escape Sequences

- Some escape sequence are like in Python:

\n – escapes the new line

System.out.println("Hello\nWorld");

Hello

World

\t – escapes the horizontal tab

System.out.println("Hello\tWorld");

Hello World

\\ - escapes the backslash itself

System.out.println("Campus\\B00000012");

Campus\B00000012

Your Turn …

- What is the output of each of the following blocks of code? Do they compile? Do they give errors when you try to run them?

1. System.out.printf("Price:%10.2f\n", 25.33333);
2. System.out.printf("Price:%.2f\n", “25.33333”);
3. System.out.printf("Price:%n%d\n", 25.33333);
4. System.out.printf("Price:%nd\n", 25.33333);
5. String name = “Anna”; int age = 5;
   System.out.printf(“Hello %s, you are %d today", name, age);
6. System.out.printf("hello world");
7. System.out.println(“We need to use the \\ to escape some special characters”);
8. System.out.printf(“Hello“ + “\tWorld”);
9. System.out.printf("hello %cworld", '@');
10. System.out.print("*\n**\n***\n");

Input from keyboard and import statements

- We can make our programs more flexible by asking a user for input, rather than having fixed values.
- To do so we can use the **Scanner** class.
- To use the Scanner class wee need to import it first using an **import statement**, because it is located in a different package: **java.util** package
- The only package that does not need to be imported because it is automatically loaded by the JVM is the **java.lang** package.
- You can import one class/member at a time:

```
import java.util.Scanner; // best practice
```

- Or you can import the whole package (all the members will be imported):

```
import java.util.*;
```

More on import statements and packages…

- Classes are grouped into packages—named groups of related classes—and are collectively referred to as the Java class library, or the **Java Application Programming Interface** (Java API).
- In the Java API (similar to the Python Standard library) you can find a rich set of predefined classes that you can reuse rather than "reinventing the wheel."
- You can create your own package using the following declaration syntax: **package nameOfPackage;**
- **The package declaration must be the first line of code** that is read by the compiler;
- If you don't specify a package, **the no-name package** will be assigned to your class (which we will use for now).
- Similar to Python, we use import declarations to identify the predefined classes used in a Java program.
- The import statement helps the compiler locate a class that is used in this program.

Input from keyboard with Scanner

- Once imported, the Scanner class can then be used to open an input stream as follows:

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

Reference Data type (about which we will learn later)

The name of the variable; it can be scanner, or myInput, or in, etc.

Creating a new object of type Scanner (again we will learn later).

- Scanner enables a program to read data that can come from many sources, such as the user at the keyboard or a file on disk.
- The new keyword creates an object.
- Standard input object, System.in, enables applications to read bytes of information typed by the user.
- Scanner object translates these bytes into types that can be used in a program.

Input from keyboard with Scanner

- Next, we should (although is not a compilation necessity) prompt the user to enter the input, using a print statement; for instance, if we want the user to enter a number, we can prompt him as follow:

  System.out.print ("Enter a number: ");

- Then you can declare a suitable variable to store the input as follows:

  int number = input.nextInt(); /* reads a string from the keyboard, then tries to converts it to an integer and, if successful, assign it to the variable **number** */

Include this line so you can use the Scanner class.

Create a Scanner object to read keyboard input.

Display a prompt in the console window.

Define a variable to hold the input value.

```
import java.util.Scanner;
.
.
Scanner in = new Scanner(System.in);
.
.
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Don't use println here.

The program waits for user input, then places the input into the variable.

Useful Scanner class methods

- To get a long value use **nextLong():**

  long n1 = input.nextLong();

- To get an double value use **nextDouble():**

  double n2 = input.nextDouble();

- To get an float value use **nextFloat():**

  float n3 = input.nextFloat();

NOTE: All methods designed for numerical types will throw an exception (a kind of error) if the input string cannot be converted to the required datatype:

EXAMPLE: int n = input.nextInt(); /* if the user enters a whole number, the program works fine, but if the user enter something else, such as abc or 3.7, the program will end with a java.util.InputMismatchException*/

- To get a String value use **next()** or **nextLine():**

  String str = input.next (); or **String str = input.nextLine();**

NOTE: next() returns a token (such as a word separated by a space), whereas nexLine() returns the entire line (which may include several tokens); nextLine can also buffer all the input until it finds a new line character.

```java
import java.util.Scanner;

/**
 * demonstrates the use of Scanner methods
 */

/**
 * @author Aurelia Power
 *
 */
public class ScannerDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // create the scanner object to read the input
        Scanner in = new Scanner(System.in);
        // prompt the user to enter input
        System.out.print("What is your name? ");
        //take input and store it into an appropriate variable
        String name = in.nextLine();
        //prompt again
        System.out.print("How old are you? ");
        //take input again and store it into a variable
        int age = in.nextInt();
        // do something with the data inputted
        System.out.printf("Hello %s, I believe you are %d\n", name, age);
    }
}
```

Sample output:

```
What is your name? Aura
How old are you? 25
Hello Aura, I believe you are 25
```

Other Useful Scanner class methods

- The Scanner class also has methods to **check for all input** data types (we look at 3 only):

- public boolean hasNextInt() checks whether the next token in the input is a valid can be interpreted as an int;
- public boolean hasNextDouble() which checks whether the next token in the input can be interpreted as a double;
- public boolean hasNextBoolean() which checks whether the next token can be interpreted as a boolean;

- All these methods return a Boolean value so they can be used as conditions
- These methods do NOT advance the scanner like the versions of the next methods.

EXAMPLE: **if(! in.hasNextInt()){ // assume all declarations are valid**

```
    System.out.println("Cannot interpret the input as an integer value…");
}
```

Your Turn…

- **Will the following statements work? (assume imports and in already declared and created)**

```
double unitPrice = in.nextInt();
int quantity  = in.nextInt();
```

- **What is problematic about the following sequence?**

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextDouble();
Scanner in = new Scanner(System.in);
```

- **What is problematic about the following sequence?**

```
 Scanner sc = new Scanner(System.in);
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextDouble();
```

- Using the printf method, print the values below so that the output looks something like this, that is, the numbers should line up to the right:

```
Quantity:        10
Unit Price:    25.3
```

Arithmetic Operators

- Java has **four familiar arithmetic operators**, plus a remainder operator for this purpose (we also learnt them last semester in Python).

The arithmetic operators of Java	
Operation	**Java operator**
addition	+
subtraction	-
multiplication	*
division	/
remainder	%

- **REMEMBER:** the + operator also acts as a concatenation operator when used with Strings;
- The + operator will carry out mathematical operations when applied to numerical values.

Division, Integer division, and remainder

- Division works like in maths, as long as one of the operands is a floating-point number:

```
System.out.println(7.0 / 4.0); // 1.75
System.out.println(7 / 4.0); // 1.75
System.out.println(7.0 / 4); // 1.75
```

- But, if both operands are integers, then the result will always be an integer, and anything after decimal point will be discarded:

```
System.out.println(7/ 4); // 1 – here the remainder 3 is discarded
System.out.println(25/ 5); // 5
System.out.println(5/ 2); // 2 – here the remainder 1 is discarded
```

Integer division and remainder

- If we want to find out what the remainder of the division between 2 integers, we can use the **modulus operator (%)** which returns the remainder of ***integer division*** **(like in Python).**

Examples of the modulus operator in Java	
Expression	**Value**
29 % 9	2
36 % 5	1
6 % 8	6
40 % 40	0
10 % 2	0

Arithmetic Expressions and Operators Precedence

- **Works exactly like in Python.**
- **Complex mathematical computations can be written using arithmetic expressions;**
- Parentheses are used in Java expressions in the same manner as in algebraic expressions. For example, to multiply **a** times **b + c** we write: **a * (b + c)**
- **Like in maths, certain operations take precedence over others in an arithmetic expression :**

  - parentheses are evaluated first

  - multiplication, division, and/or remainder are carried out next

  - then, addition and/or subtraction
- **NOTE:** To carry out addition/subtraction before multiplication/division/remainder, you must use parentheses.

Operators Precedence Rules:

- Java applies the operators in arithmetic expressions in a precise sequence determined by the **rules of operator precedence:**

1. Operators in expressions contained within **pairs of parentheses are evaluated first.**
   - This can therefore be used as a mechanism to force an order of evaluation in your program.
   - In cases of nested or embedded parentheses, the operators in the innermost pair are applied first.

2. **Multiplication, division and modulus** operations are applied next, **from left to right**.

3. **Addition and subtraction operations** are applied last **from left to right.**

Your Turn…

- **What does each of the following statements output?**

```
System.out.println(10 / 4);        System.out.println(3.0 / 2);
System.out.println(1 / 2);         System.out.println(1 % 2);
```

- **What is the value of**

  - 1729 / 10
  - 1729 % 10

- **What is the value of variable a after executing all the following code?**

```
int a = 37;  a = a + 5/2; int b = 3+4*3/5; a = b; a = a + 2;
```

- **What does each of the following statements output?**

```
System.out.println(4 + (5 * (6 / 3)));        System.out.println(4 + ((5 * 6) / 3));
System.out.println(1 + 2 - (3 * (4/5)));      System.out.println(1 % 2 - ((3 * 4) / 5));
```

Relational and Equality Operators

- Many times in our programs we need to compare values;
- Comparisons always yield boolean values: either true or false

EXAMPLES:

- we want to see if age is smaller than 18: age <18
- check to see if the amount spent on 10 coffees is the same as the pocket money allowed for the week: amountSpent == pocketMoney
- Check to see if I passed an exam: myMark != 'F' (**F is for fail**)
- For all the above examples, because we deal with primitive data types, we can use the **relational operators** that Java has for comparing primitive data types.

NOTE: to compare if 2 Strings are equal we use the *equals method*, because Strings are not primitive data types:

```
System.out.println("Anna".equals("Hellen")); // outputs: false
```

Relational and Equality Operators in Java

Java	Math Notation	Description
>	$>$	Greater than
>=	$\geq$	Greater than or equal
<	$<$	Less than
<=	$\leq$	Less than or equal
==	$=$	Equal
!=	$\neq$	Not equal

Expression	Value	Comment
3 <= 4	true	3 is less than 4; <= tests for “less than or equal”
3 =< 4	Error	The “less than or equal” operator is <=, not =<. The “less than” symbol comes first
3 > 4	false	> is the opposite of <=
4 < 4	false	The left hand side must be strictly smaller than the righthand side
4 <= 4	true	Both sides are equal; <= tests for “less than or equal”
3 == 5 - 2	true	== tests for equality
3 != 5 - 1	true	!= tests for inequality. It is true that 3 is not 5 – 1
3 = 6 / 2	Error	Use == to test for equality, not = which is the assignment operator
"10" > 5	Error	You cannot compare a string to a number;
/*assume s1 is “ab”and s2 is “aB”;*/ s1.equals(s2)	false	Always use the equals method to check whether two strings have the same sequence of characters.
/*assume s1 is “ab”and s2 is “aB”;*/ !s1.equals(s2)	true	! Negates the value of the expression s1.equals(s2)

Precedence of Arithmetic and Relational Operators

- Note that the **arithmetic operators have higher precedence** than the relational operators → mathematical operations are carried out before the comparisons;
- So far, among what we have learnt …

1. Arithmetic operators are evaluated first
2. Relational operators are evaluated next
3. The assignment operator is evaluated last.

NOTE: putting spaces between component characters of some relational/equality operators is a syntax error:

= = - will not compile, but == will

< = - will not compile, but <= will

> = - will not compile, but >= will

Program Example

```java
public static void main(String[] args) {
    System.out.println(3 > 4); //false
    System.out.println(3 >= 4); //false
    System.out.println(3 < 4); //true
    System.out.println(3 <= 4); //true
    System.out.println(3 + 10 > 4); //true
    System.out.println(3 - 2 > 4); //false
    int num1 = 34;
    int num2 = 25;
    System.out.println(num1 < num2); //false
    System.out.println(num1 > num2); //true
    System.out.println(num1 >= num2 + 9); //true
    char a = 'a';
    System.out.println(a == 'a'); //true
    System.out.println(a == 'A'); //false
    String name = "Aurelia";
    String name2 = "Aura";
    System.out.println(name.equals("Aurelia")); //true
    System.out.println(name.equals(name2)); //false
    System.out.println(!name.equals(name2)); //true
}
```

Don't forget class declaration!!

Your Turn…

- Which of the following conditions are true, provided a is 3 and b is 4?
  - **a + 1 <= b**
  - **a + 1 >= b**
  - **a + 1 != b**
  - **a + 1 = b**
- What is the output after executing all the following statements?

```
String message1 = “hello\nworld”;
String message2 = “hello ” + “\nworld”;
System.out.println(message1.equals(message2));
```

Control Structures

Like in Python, all Java programs are written in terms of **three control structures**:

- Sequence structure
- Selection structure
- Repetition structure.

- Every program is formed by using one, two or all control structures: the sequence statement, selection statements (three types) and repetition statements (three types), depending on the problem the program implements.

Sequence Control Structure

- Sequence structure is built into Java same as in Python → unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written.
- Java lets you have as many actions as you want in a sequence structure.

EXAMPLE: (assume class and main method declarations, and Scanner object instantiated): all statements below are executed in order in which the appear

```
System.out.prin("Enter your age: ");
int age = in.nextInt();
System.out.println("You are " + age + " years old");
System.out.println(" Nice !!");
```

The Java syntax for single selection – the if statement

- This form of selection is called **single selection** in Java because it involves only one choice;
- It makes use of a **boolean condition – the guard**.
- If the boolean condition is **true** then the program branches out to execute the instructions that correspond to that course of action;
- If the boolean condition is **false**, then the program ignores the instructions associated with that course of action (it skips them), and moves to the next statement.

```
if ( /* boolean condition goes here */ )
{
   // conditional instruction(s)/statement(s) go here
}
```

Example

```java
public class SingleSelectionDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // instantiate the Scanner object
        Scanner in = new Scanner(System.in);
        //prompt user to enter age
        System.out.print("Enter your age:  ");
        //store the input into a variable called age
        int age = in.nextInt();
        // output message containing age
        System.out.println("You are " + age + " years old");
        /* output message stating that user is an adult only
        if the age is equal to or over 18 */
        if( age >= 18){
            System.out.println("You are an adult");
        }
        // output a nice message
        System.out.println(" Nice !!");
        //close the Scanner
        in.close();
    }
}
```

Don't forget to import the Scanner class…

<u>Possible output for when age is greter or equal to18</u>

Enter your age: 25
You are 25 years old
You are an adult…
Nice !!

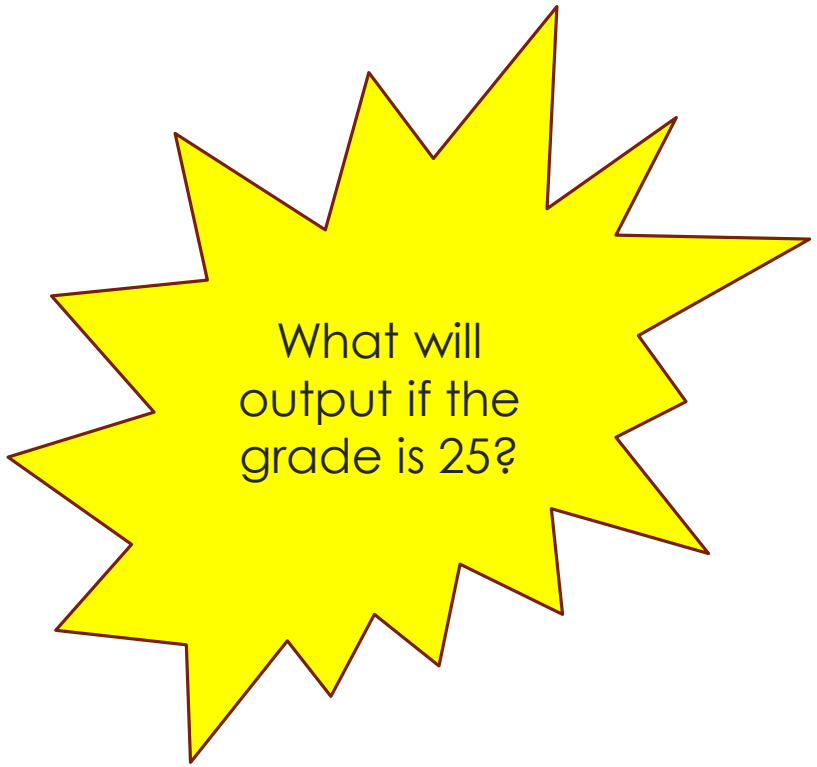
<u>Possible output for when age is smaller than 18</u>

Enter your age: 12
You are 12 years old
Nice !!

Multiple if statements

- Like in Python, when using multiple **if** statements we must be careful of how we formulate our boolean conditions,:
- For instance, we want to print different messages for different grades:

```
if (grade < 80){
  System.out.println("it's  a B");
}
if (grade < 60){
  System.out.println("it's  a C");
}
if (grade < 40){
  System.out.println("it's  a D");
}
if (grade < 35){
  System.out.println("it's  an F");
}
```



Conditional operators in Java

- We can combine boolean conditions using conditional AND (&&) and conditional OR (||);
- Java has a third conditional operator – the ternary operator

So… there are <u>3 conditional operators</u>:

1. && - Conditional AND
2. || - Conditional OR

These operators take 2 boolean operands and exhibit "short-circuiting" behavior like the **and** and **or** in Python, which means that the second operand is evaluated only if needed.

3. ? : – the ternary operator takes 3 operands (the first is a boolean) and is really a shorthand for an if-else statement.

```
public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2)){
                System.out.println("value1 is 1 AND value2 is 2");
        }
        if((value1 == 1) || (value2 == 1)){
                System.out.println("value1 is 1 OR value2 is 1");
        }
}
```

```
public static void main(String[] args){
        String s1 = “it’s true”;
        String s2 = “it’s false”;
        boolean condition = true;
        String result = condition ? s1 : s2;
        System.out.println(result);
}
```

Multiple if statements

- We can formulate our boolean conditions more precisely so that they specify both ends of a range, using **relational operators** and the **AND** (&&) **conditional operator**;
- For instance:
  - **A** is between 80 and 100 inclusive;
  - **B** is between 60 and 79 inclusive;
  - **C** is between 40 and 59 inclusive;
  - **D** is between 35 and 39 inclusive;
  - **F** is anything below 35;

```
if (grade >= 80 && grade <= 100){
   System.out.println("it's  an A");
}
if (grade >= 60 && grade < 80){
   System.out.println("it's  a B");
}
if(grade >= 40 && grade < 60){
   System.out.println("it's  a C");
}
if (grade >= 35 && grade < 40 ){
   System.out.println("it's  a D");
}
if (grade < 35 && grade > =0){
   System.out.println("it's  an F");
}
```

What will output if the grade is 25?

Logical operators in Java

- The conditional AND (&&) and the conditional OR (||) are 2 of the **logical operators** that can be used in Java
- Like in Python:
  - conditional AND (&&) – evaluates to true only when both operands are true
  - conditional OR (||) – evaluates to true when at least one of the conditions is true

- There are **other logical operators** in Java:
  - Exclusive OR (^) – evaluates to true only when one of the operands is true
  - Negation/NOT (!) – is a unary operator that simply negates the truth of its operand (acts the same as the **not** in Python)

REMEMBER: &&, || and ^ are binary operators because they take 2 operands, whereas ! is a unary operator because it takes only one operand.

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator (see Appendix B).
`0 < x < 100`	**Error**	**Error:** This expression does not test whether `x` is between 0 and 100. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer 100.
`x && y > 0`	**Error**	**Error:** This expression does not test whether `x` and `y` are positive. The left-hand side of `&&` is an integer, `x`, and the right-hand side, `y > 0`, is a Boolean value. You cannot use `&&` with an integer argument.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Your turn…

- Consider the following code blocks (assume previous valid declarations for class and main); what do they output?

```
int x = 3+4; int y = 3 +x;
System.out.println(x > y ? “x > y”: “x < y”);

String s1 = “FOP2”; String s2 = “FOP1”;
System.out.println( (s1.equals(s2) ^ 3 < 4) ? “^=^” : “v=v”);
```

- Assume previous valid declarations and x=2, y =7, z = 1; what is the output of the following code block?

```
if((x >= 100 || y == 7) && (y == z + 6) ){
        System.out.println("python is better");
} if((x != 100 ^ y == 7) || !(“python”.equals(“java”)) ){
        System.out.println("java is better");
}
```

The Java syntax for double selection – the if…else statement

- This form of selection is called **double selection** in Java because it involves only two courses of action (it acts exactly like in Python in fact, only the syntax is different);
- It also makes use of a **boolean condition – the guard**.
- If the boolean condition is **true** then the program branches out to execute the instructions that correspond to that course of action;
- If the boolean condition is **false**, then the program branches out to execute the instructions associated with the other course of action.
- It will never execute both branches, only one.

```
if ( /* boolean condition goes here */ )
{
   // conditional instruction(s)/statement(s) go here
}
else
{
    // conditional instruction(s)/statement(s) go here
}
```

Re-writing the age program

```java
public class DoubleSelectionDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // instantiate the Scanner object
        Scanner in = new Scanner(System.in);
        //prompt user to enter age
        System.out.print("Enter your age:  ");
        //store the input into a variable called age
        int age = in.nextInt();
        // output message containing age
        System.out.println("You are " + age + " years old");
        /* output message stating that user is an adult only
        if the age is equal to or over 18 */
        if( age >= 18){
            System.out.println("You are an adult...");
        }
        /* output message stating that user is under age
        if the age is smaller than 18 */
        else{
            System.out.println("You are under age...");
        }
        // output a nice message
        System.out.println(" Nice !!");
        //close the Scanner
        in.close();
    }
}
```

Don't forget to import the Scanner class…

<u>Possible output for when age is greter or equal to18</u>

your age: 25
You are 25 years old
You are an adult...
Nice !!

<u>Possible output for when age is smaller than 18</u>

Enter your age: 12
You are 12 years old
You are under age...
Nice !!

Multiple if-else statements

- We can also have a series of if-else statements like in Python (except the keyword is not elif but else if, and the syntax is a bit different).
- You can have **as many else if clauses as needed**;
- The **last else clause is optional**;
- Only **one branch (one set of instructions) will be executed**, depending on which condition evaluates to true;
- The **order** in which you place them also matters, unless the boolean conditions are very specific.

```
if ( /* boolean condition goes here */ ){
  // conditional instruction(s)/statement(s) go here
}
else if ( /* another boolean condition goes here */ ){
      // conditional instruction(s)/statement(s) go here
}
.
.
.
else{
      // conditional instruction(s)/statement(s) go here
}
```

Your turn…

1. Consider the following if statement to compute a discounted price:

```
if (originalPrice > 100) {
    discountedPrice = originalPrice - 20;
} else {
    discountedPrice = originalPrice - 10;
}
```

What is the discounted price if the original price is **95**? **100**? **105**?

2. What is the output of the following statement if **rainy** is true?

```
System.out.println(rainy : "stay in" ? "go out");
```

3. Assume **int a = 3;** what will the next block of code output?

```
if(a == 3) { System.out.println("a equals to 3"); }
if(a == 2+1) { System.out.println("a equals to 2+1"); }
if(a+1 == 2+1) { System.out.println("a  + 1 equals to 2 + 1"); }
if(a+1 >= 3) { System.out.println("a + 1 greater or equal 3"); }
```

4. Write a conditional statement with three branches that sets **n** to 1 if x is positive, to –1 if x is negative, and to 0 if x is zero.

Your turn…

1. What is the value of each variable after the if statement?

- int n = 1; int k = 2; int r = n; if (k < n) { r = k; }
- int n = 1; int k = 2; int r; if (n < k) { r = k; } else { r = k + n; }
- int n = 1; int k = 2; int r = k; if (r < k) { n = r; } else { k = n; }
- int n = 1; int k = 2; int r = 3; if (r < n + k) { r = 2 * n; } else { k = 2 * r; }

2. Find the errors (if any) in the following if statements:

- if x > 0 then System.out.print(x);
- x = in.nextInt(); if(in.hasNextInt()){ sum = sum + x; } else {System.out.println("Bad input for x"); }
- String letterGrade = "F"; if (grade >= 90) { letterGrade = "A"; } if (grade >= 80) { letterGrade = "B"; } if (grade >= 70) { letterGrade = "C"; } if (grade >= 60) { letterGrade = "D"; }

3. What do these code fragments print?

- int n = 1; int m = -1; if (n < -m) { System.out.print(n); } else { System.out.print(m); }
- int n = 1; int m = -1; if (-n >= m) { System.out.print(n); } else { System.out.print(m); }