ICS 102 Lab02 – Arithmetic Expressions and Output Formatting

**Objectives:** Introduction to:

* Java keywords and identifiers
* Arithmetic operators and expressions
* Mixed-mode arithmetic and automatic type conversions
* Type casting
* The Math class
* Output Formatting using format specifiers

# Primitive data types

All data in Java falls into one of two categories: **primitive data** and **objects**.

* A primitive data value uses a small, fixed number of bytes.
* There are only eight primitive data types (Example: int, double).
* A programmer cannot create new primitive data types.

**2. Keywords and Identifiers**

An identifier is the name of an item in a program (e.g., variables, named constants, method names, class names etc.).

**Java is a free form, case-sensitive language.**

Identifier naming rules in Java are:

* The first character of an identifier must be a letter, underscore, or $
* All the other characters of an identifier must be letters, digits, or underscores
* An identifier may be arbitrarily long (A compiler may impose a limit on the length of an identifier)
* The name of an identifier cannot be one of the words that the Java language has reserved for its own use. A reserved word is also called a keyword. This means that you cannot use one of the Java keywords to name your identifiers.
* Some keywords: **int**, **double**, **float**, **new**, **void**, **if**, **else**, **switch**, **do**, **while**, **for** [See complete list in Appendix 1 on page 1199 of your textbook: Absolute Java 6th Edition]

Java has some pre-defined words or **Standard Identifiers** that are not keywords. You should not use these pre-defined words to name your identifiers to avoid confusion. Examples are names of standard classes and names of standard methods: **System**, **Math**, **Scanner**, **String**, **print**, **println**, **printf** etc.

# 3. Declaring and using variables and final variables [defined constants].

A variable declaration is a request to the system to allocate memory for data values to be stored. It involves specifying the type and a name for the variable. The type is used by the system to know how many bytes of memory to allocate. For a programmer to refer to the allocated memory, he uses the name.

|  |  |
| --- | --- |
| e.g. int age;  int numberOfStudents;  double average; | lab01_1a.gif |

If you have an initial value to assign to your variables, you can do so at the point of declaration as follows:

|  |  |
| --- | --- |
| int numberOfStudents = 25; | lab01_1b.gif |

# Within a method, declaration statements can appear anywhere in the program before their variables are first used.

A single declaration may be used to declare more than one variable:

int numStudents, numCourses, count = 0;

such a declaration may spread over several lines:

int numStudents,

numCourses,

count = 0;

A final variable is a variable that can be initialized only one time. To declare a variable as final, add the keyword **final** to the declaration.

Examples:

final int NUMBER\_OF\_HOURS\_IN\_A\_DAY = 24;

final double MAX\_SPEED = 120.0;

**3.1 The scope of a local variable**

In Java, matching curly brackets { } define a block. A variable that is declared within a method block is called a local variable.

The scope of a variable is the part of the program over which the variable name can used. You cannot use a local variable before its declaration; otherwise the compiler issues a compilation error: **“error: cannot find symbol”**

The scope of a local variable is from the point it is declared to the end of the block in which it is declared. Statements outside this region cannot access such a local variable.

A local variable must be initialized (i.e., it must have a value) before it can be used in an expression; otherwise the compiler issues a compilation error: **“error: variable might not have been initialized”**

You cannot declare two variables with the same name in the same scope; otherwise the compiler issues a compilation error: **“error: variable already defined in method”**

**4. Statements**

* In programming, a statement is an instruction to a computer to do something.
* A statement forms part of the sequence of program execution.
* **In Java, every simple statement is terminated with a semicolon.** Example:

System.out.println("ICS 102 Lab02");

* Multiple simple statements can be written on a single line. Example:

System.out.println("ICS 102 Lab02"); System.out.println("Java ");

* In Java, an empty statement is legal; it does nothing:

**;**

**4. 1. Assignment Statements**

We can change the value of a variable any number of times within a program. One-way of assigning or changing the value of a variable is using the assignment statement, which has the form:

*variable = expression;*

Example:

int first, second, count = 0;

double average;

first = 10;

second = 20;

average = (first + second)/2.0;

count = count + 1;

**4.2. Multiple assignments in a single statement**

Java supports multiple assignments in a single statement. Example:

**int** a, b, c;   
a = b = c = 250;

**Note:** The code below will produce a compilation error: **“error: cannot find symbol”**

int a = b = c = 250;

You cannot use multiple assignments in a declaration statement.

**4.3. Compound Assignment Operators**

In programs, it is quite common to get the current value of a variable, modify it, and then assign the resulting value back to the original variable.

Example:

**x = x + 10;**

This statement specifies that the value 10 is to be added to the current contents of the variable ***x*** and the resulting value becomes the new contents of ***x***. Effectively, this statement increases the value of the variable ***x*** by 10. Java provides a shorthand way to write this statement using the compound assignment operator: **+=** .Instead of writing the statement:

**x = x + 10;**

We can write:

**x += 10;**

In general, in Java an assignment of the form:

**variableX = variableX operator Expression ;**

Can be written in the form:

**variableX operator= Expression;**

For arithmetic operators, the compound assignment operators are \*= , /= , %= , +=, and -=. These operators have the same priority and associativity as the assignment operator =

Note: Sometimes it is not possible to make the previous two assignment forms equivalent. Example: **int x = 5, y = 2;  
 x = x \* 5 + y;  
 System.out.println(x); // output: 27  
 x = 5;  
 x \*= 5 + y;  
 System.out.println(x); // output: 35**

However the forms:

**variableX = variableX operator (Expression);**

**variableX operator= (Expression);**

are always equivalent.

**5. String concatenation operator**

The operator **+** when applied to two strings creates a new **String** object that is the concatenation of the two **String** objects. The operator may also be applied to a **String** and a numeric expression to produce a **String** object that concatenates the **String** and the expression. The **String** concatenation operator has the same priority as the addition and subtraction operators.

Example: The code:

**String mystring = "KFUPM";**

**System.out.println("I am a student of " + mystring);**

**int num1 = 12, num2 = 45;**

**System.out.println(num1 + " + " + num2 + " = " + (num1 + num2));**

produces the output:

I am a student of KFUPM

12 + 45 = 57

# Examples

The following program assigns two values and prints the following

* Their sum
* Their difference
* Their product
* Their average

**/\* computes the sum, difference, product of two integer numbers \*/**

**public class Arithmetic**

**{**

**public static void main(String[] args) {**

**int num1 = 15;**

**int num2 = 19;**

**System.out.println("The sum is: " + (num1 + num2));**

**System.out.println("The difference is: " + (num1 - num2));**

**System.out.println("The product is: " + (num1 \* num2));**

**System.out.println("The average is: " + ((num1 + num2)/2.0));**

**}**

**}**

**Note:** By introducing additional variables, the previous program may be written as:

**/\* computes the sum, difference, product of two integer numbers \*/**

**public class Arithmetic**

**{**

**public static void main(String[] args) {**

**int num1 = 15 , num2 = 19, sum, diff, product;**

**double average;**

**sum = num1 + num2;**

**diff = num1 – num2;**

**product = num1 \* num2;**

**average = (num1 + num2) / 2.0;**

**System.out.println("The sum is: " + sum);**

**System.out.println("The difference is: " + diff);**

**System.out.println("The product is: " + product);**

**System.out.println("The average is: " + average);**

**}**

**}**

1. **Increment and Decrement operators**

Each of the post-increment and pre-increment operators increment the value of a variable by one:

|  |  |
| --- | --- |
| variable++ | post-increment operator |
| ++variable | pre-increment operator |

Each of the post-decrement and pre-decrement operators decrement the value of a variable by one:

|  |  |
| --- | --- |
| variable-- | post-decrement operator |
| --variable | pre-decrement operator |

The increment operators have similar effect if they appear alone in an expression. Example:

**int x = 3;**

**x++; // x is incremented to 4**

**x = 3;**

**++x; // x is incremented to 4**

If these operators appear in an assignment statement or in an expression in which they are combined with other operands and operators, then for the pre-increment operator, the variable is first incremented and then this incremented value is used to evaluate the expression. For post-increment operator, the expression is first evaluated with the current value of the variable, then the variable is incremented. The decrement operators behave in a similar way, except that they decrement the variable:

Examples:

|  |  |
| --- | --- |
| Post-Increment | Pre-Increment |
| **int n = 8;**  **int result = n++;**  System.out.println("n = " + n + ", result = " + result);  Output: n = 9, result = 8 | **int n = 8;**  **int result = ++n;**  System.out.println("n = " + n + ", result = " + result);  Output: n = 9, result = 9 |
| double x = 2.0, y; y = x++ \* 5.0; System.out.println("x = " + x + ", y = " + y);  Output: x = 3.0 , y = 10.0 | double x = 2.0, y; y = ++x \* 5.0; System.out.println("x = " + x + ", y = " + y);  Output: x = 3.0 , y = 15.0 |

|  |  |
| --- | --- |
| Post-decrement | Pre-decrement |
| **int n = 8;**  **int result = n--;**  System.out.println("n = " + n + ", result = " + result);  Output: n = 7, result = 8 | **int n = 8;**  **int result = --n;**  System.out.println("n = " + n + ", result = " + result);  Output: n = 7, result = 7 |
| double x = 4.0, y; y = x-- \* 5.0; System.out.println("x = " + x + ", y = " + y);  Output: x = 3.0 , y = 20.0 | double x = 4.0, y; y = --x \* 5.0; System.out.println("x = " + x + ", y = " + y);  Output: x = 3.0 , y = 15.0 |

1. **Integer division**

The result of an expression in which all the operands are integers is an integer.

|  |  |
| --- | --- |
| Expression | Result |
| 10 / 2 | 5 |
| 7 / 3 | 2 |
| 4 / 5 | 0 |
| 6 + 2 \* 3 | 12 |

You have to be careful when using integer division as sometimes it may introduce logic errors in your program. Example:

Consider the following program fragment that computes the average of two integers:

**int num1 = 2, num2 = 5, average;**

**average = (num1 + num2 ) / 2;**

**System.out.println("Average = " + average);**

The output is:

Average = 3

which is not the average of 2 and 5.

**The modulus operator %**

In Java, the modulus operator may have floating point operands. In ICS 102 we will only consider this operator with positive integer operands. When applied to two integers the result is the remainder of dividing the integers. Examples:

|  |  |
| --- | --- |
| Expression | Result |
| 7 % 2 | 1 |
| 12 % 2 | 0 |
| 8 % 3 | 2 |
| 4 % 7 | 4 |
| 9 % 12 | 9 |

**Example:** Write a Java Program fragment that converts 34 days into weeks and days:

Solution: There are 7 days in one week, hence:

**int days, weeks, givenNumDays = 34;**

**int final NUM\_DAYS\_IN\_WEEK = 7;   
weeks = givenNumDays / NUM\_DAYS\_IN\_WEEK;  
days = givenNumDays % NUM\_DAYS\_IN\_WEEK;  
System.out.println("There are " + weeks + " weeks and " + days + " days in " + givenNumDays + " days");**

1. **Mixed-mode arithmetic operations and Mixed-mode assignments**
   1. **Automatic type conversions**

If an arithmetic expression contains operands of different primitive types, then an operand of a lower type is automatically converted to a higher type. A conversion from a lower type to a higher type is called a **widening conversion**:

|  |
| --- |
| Low type High type |
| byte 🡪 short 🡪 int 🡪 long 🡪 float 🡪 double |

The result of the expression is that of the higher type. When values of type byte and short are used in arithmetic expressions, they are temporarily converted

to type int.

An expression of a lower type may be assigned to a variable of a higher type. The following are all valid assignment statements:

**int x = 3;**

**long z = x \* 2;**

**float y = z + 4L;**

**double w = y \* 2;**

We can use mixed-mode arithmetic in the example of finding the average of two integers:

**int num1 = 2, num2 = 5;**

**double average;**

**average = (num1 + num2 ) / 2.0;**

**System.out.println("Average = " + average);**

The output is:

Average = 3.5

* 1. **Type casting**

Java primitive data types arranged from low to high:

(low) byte 🡪 short 🡪 char 🡪 int 🡪 long 🡪 float 🡪 double (high)

In Java, you **cannot** assign the value of an expression of a higher type to a variable of a lower type. Such type of conversions are called narrowing conversions. The exceptions to this rule are:

* A **byte** variable can be assigned a constant integer in the range -128 to 127 inclusive without type casting [by default integer numbers are of type int]
* A **short** variable can be assigned a constant integer in the range -32768 to 32767 inclusive without type casting
* A **char** variable can be assigned a constant integer in the range 0 to 65535 inclusive without type casting

Note: A type cast is required if the assigned value is in a variable:

int num1 = 32767;

short num = (short) num1;

The following are all invalid assignment statements:

**int num2 = 45L;**

**int num3 = 3 \* 2.0;**

**float num4 = 45.7; // by default 45.7 is of type double**

Each one of the above will produce a compilation error: **“error: possible loss of precision”**

Value casting or type casting consists of converting a value of one type into a value of another type. For example, you may have an integer value and you may want that value in an expression that expects a short number. Value casting is also referred to as explicit conversion.

To cast a value or an expression, precede it with the desired data type in parentheses. Examples:

**double num2 = 24.32;**

**int num3 = (int) num2 \* 2;**

**float num4 = 45.7F / (float) num2;**

When performing explicit conversion, you must ensure that the value being cast does not overflow or underflow. For example, if you want an integer value to be assigned to a short variable, the value must fit in 16 bits, which means it must be between -32768 and 32767. Any value beyond this range would produce an unpredictable result. Consider the following program fragment:

**int iNumber = 680044;**

**short sNumber = (short)iNumber;**

**System.out.println("Number = " + iNumber);**

**System.out.println("Number = " + sNumber);**

This would produce:

Number = 680044

Number = 24684

Notice that the result is not reasonable.

Note:

* When you convert a floating-point type to an integer type, any fractional portion is truncated (discarded). **Note**: The value is not rounded.
* The operand of the cast operator is not changed by the operation. A copy of the value is made that is of the specified data type.

# Priority and Associativity of Arithmetic Operators

Arithmetic operators, operands, and parenthesis are used to form arithmetic expressions. Evaluation starts from the inner-most parenthesis. The evaluation is according to priority and associativity rules.

**Operator Precedence**

Operator precedence determines the order in which operators are evaluated. Operators with higher precedence are evaluated first.

Example:

3 + 4 \* 5 // evaluates to 23

The multiplication operator has higher precedence than the addition operator and thus will be evaluated first.

**Associativity**

Associativity determines the order in which **adjacent** operators of the same precedence are processed. Two unary operators are adjacent if they appear one after another in an expression. Two binary operators are adjacent if they are separated by one operand in an expression.

For example, consider an expression:

**a** OP **b** OP **c**

Left-associativity (left-to-right) means that it is processed as (**a** OP **b**) OP **c**, while right-associativity (right-to-left) means it is interpreted as **a** OP (**b** OP **c**).

Priority and Associativity Rules:

* Expressions inside parentheses are evaluated first
* Nested parentheses are evaluated from the innermost to the outermost parentheses.
* Operators higher in the chart have higher precedence (higher priority).
* Operators in the same row in the chart have equal precedence. If these operators are adjacent in an expression, their associativity determines which operator is evaluated first; otherwise the evaluation is from left to right.
* Some operators are not associative: for example **x++--** is invalid.
* Parentheses may be used to control the order of evaluation.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Description** | **Operators** | **Associativity** |
| High Priority  Low Priority | Parentheses and method calls | ( ) | Left to Right |
|  | ++ (post-increment) -- (post-decrement) | Not associative |
|  | + (unary) - (unary) ++ (pre-increment) --(pre-decrement) | Right to left |
| Type casting | (type) | Right to left |
| multiplication, division, mod | \* / % | Left to Right |
| addition, subtraction, String concatenation | + - + | Left to Right |
| assignment | = += -= \*= /= %= | Right to Left |

* Java does not have an exponentiation operator. To raise a number to a power we can use the ***pow***method of the ***Math*** class. Example **x5**  is written as **Math.pow(x, 5)**

Note: Parentheses are used to change the normal order of evaluation. For example, if you wanted to code the following algebraic expression in Java:

You could write:

y = (4 \* a \* c – d) / (2 – b);

Without the parentheses, the equation implemented would be:

– b

which in Java corresponds to: y = 4\*a\*c – (d/2) – b;

**Expressions** are formed by combining initialized variables, constants, and method return values using operators.

Each of the following is an expression:

* + - 1. A literal constant
      2. The name of an initialized constant [final variables]
      3. An initialized variable
      4. A call to a method that returns a single value
      5. A proper combination of any of 1, 2, 3, and 4 using operators.

1. **The Math class**

To assist you with various types of calculations, the **java.lang** package contains a class named **Math**. In this class are the most commonly needed mathematical methods and constants:

|  |  |  |  |
| --- | --- | --- | --- |
| Constant or Method |  | Comment | Method return type or type of constant |
| E | *e* | e, the base of the natural logarithms. | double |
| PI | π |  | double |
| abs(x) | |x| | Absolute value of x. | double, int, float, or long depending on type of x |
| sqrt(x) |  | Square root of x. | double |
| [cbrt](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#cbrt(double))(x) |  | Cube root of x. | double |
| sin(x) |  | Sine of angle x, x is in radians. | double |
| cos(x) |  | Cosine of angle x, x is in radians. | double |
| tan(x) |  | Tangent of angle, x is in radians. | double |
| toDegrees(angrad) |  | Converts an angle in radians to the equivalent angle in degrees. | double |
| toRadians(angdeg) |  | Converts an angle in degrees to the equivalent angle in radians. | double |
| [asin](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#asin(double))(x) | sin -1 x | arc sine of a value x; the returned angle is in the range -*π* / 2 through *π* / 2. | double |
| [acos](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#acos(double))(x) | cos -1 x | arc cosine of a value x; the returned angle is in the range 0.0 through *π*. | double |
| [atan](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#atan(double))(x) | tan -1 x | Returns the arc tangent of a value; the returned angle is in the range -*π*/2 through *π*/2. | double |
| exp(x) |  | *e* is 2.7182... | double |
| log(x) | ln x | Returns the natural logarithm (base *e*) of x | double |
| [log10](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#log10(double))(x) | log 10 x | Returns the base 10 logarithm of x | double |
| round(x) |  | Rounds x.  Examples: round(3.4) = 3 , round(3.6) = 4 , round(-3.4) = -3, round(-3.9) = -4 | **long if x is double, int if x is float** |
| rint(x) |  | Rounds x if its decimal part ≥ 0.5; otherwise x is truncated. A negative number with decimal part ≥ 0.5 is rounded down.  Examples: rint(5.499999) = 5.0, rint(5.5) = 6.0, rint(5.8) = 6.0,  rint(-5.2) = -5.0, rint(-5.5) = -6.0 | **double** |
| [ceil](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#ceil(double))(x) | ⎡x⎤ | Returns the **smallest** double value that is **greater** than or equal to x, and is equal to a mathematical integer.  Examples: ceil(5.2) = 6.0, ceil(12.7) = 13.0, ceil(6.0) = 6.0,  ceil(-4.3) = -4.0 | double |
| [floor](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#floor(double))(x) | ⎣ x ⎦ | Returns the **largest** double value that is **less** than or equal to x and is equal to a mathematical integer.  Examples: floor(5.9) = 5.0, floor(12.3) = 12.0, floor(6.0) = 6.0, floor(-4.3) = -5.0 | double |
| **max**(x, y) |  | Maximum of x and y. | int, long, float, or double depending on the types of x and y. If x and y are of different types, the larger type is returned. |
| **min**(x, y) |  | Minimum of x and y. | int, long, float, or double depending on the types of x and y. If x and y are of different types, the larger type is returned. |
| **pow**(x, y) | xy |  | double |
| [hypot](file:///E:\122\ics102-122\Applications\jdk-7u11-apidocs\docs\api\java\lang\Math.html#hypot(double, double))(x, y) |  |  | double |
| **random**() |  | Returns a pseudo-random value in the interval [0.0, 1.0) | double |

To access any of these methods or constants, we need to prefix it with the name of the class **Math**. Examples:

// Assume all variables are initialized

double sqrtDiscriminant = Math.sqrt(b\*b – 4\*a\*c);

double root1 = (-b + sqrtDiscriminant)/ (2\*a);

double root2 = (-b - sqrtDiscriminant) / (2\*a);

double circleArea = 2 \* Math.PI \* Math.pow(radius, 2);

**Examples on the use of ceil and floor methods:**

A car park charges 3.50 Saudi Riyals for each hour or a fraction of an hour of parking. Write a Java program fragment that given the amount of parking hours (a double value), it calculates and displays the parking bill.

**final double CHARGE = 3.50;   
double hours, bill;  
System.out.println("Enter number of hours");  
//... Statement to input hours  
bill = CHARGE \* Math.ceil(hours);  
System.out.println("Bill = " + bill + " Saudi Riyals");**

An employee is paid a retirement benefit according to the actual number of complete years he has served (excluding any fractional part of the last year of service, if any). The benefit is calculated as:

**actual number of years served \* base salary at time of retirement**

Write a Java program fragment that given the number of years served (a double value) and the base salary, it calculates and displays the retirement benefit.

**double yearsServed, baseSalary, benefit;**

**System.out.println("Enter number of years and base salary");**

**//... Statement to input number of years and base salary**

**benefit = Math.floor(yearsServed) \* baseSalary;**

**System.out.println("Retirement Benefit = " + benefit + " Saudi Riyals");**

**Example on the random method:**

The random() method returns a pseudo-random number in the interval [0.0, 1.0). In other words: 0.0 <= Math.random() < 1.0. To get a number in a different range, you can perform arithmetic on the value returned by the random method. For example, to generate an integer between 0 and 9, you would write:

int number = (int)(Math.random() \* 10);

By multiplying the value by 10, the range of possible values becomes 0.0 <= number < 10.0.

1. **Output Formatting**

The **print** and **println** methods do not provide much control over the formatting of output. For example, the following code:

**int i = 2;  
 double r = Math.sqrt(i);  
 System.out.println("The square root of " + i + " is " + r);  
 double cost = 32.50;  
 System.out.println("The cost is " + cost + " Saudi Riyals");**

produces the output:

The square root of 2 is 1.4142135623730951  
 The cost is 32.5 Saudi Riyals

**The printf methods of System.out**

In addition to the **print** and **println** methods, the object **System.out**, has the **printf** for performing output. You can use **printf** anywhere in your code where you have previously been using **print** or **println**.

The **printf** method formats multiple arguments based on a ***format string***. The format string consists of a string constant embedded with ***format specifiers***. Except for the format specifiers, other characters in the format string are output unchanged. Calls to this method have the form:

**System.out.printf(FormatString , expression1, expression2, . . . , expressioN);**

For each expression there must be a corresponding format specifier (fs):

**System.out.printf(" fs1 fs2 . . . fsN " , expression1, expression2, . . . , expressioN);**

Example:

**double y = 5.6;**

**System.out.printf("x = %d, y = %f cm" , 2 \* 3, y);**

outputs: **x = 6, y = 5.600000 cm**

A format specifier has the following structure:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **%** | **Flags** | **Width** | **. Precision** | **Converter** |

where the shaded part is optional; it begins with a **%** and end with a character ***converter*** that specifies the kind of formatted output being generated:

|  |  |  |
| --- | --- | --- |
| converter | Formatted value | Comment |
| **d** | integer | Used to output byte, short, int, and long values |
| **f** | Fixed-point floating point | Used to output float and double values |
| **e** or **E** | E-notation floating point | The case of the displayed **e** is the same as that of the converter |
| **s** or **S** | String | If **s** is used the string is displayed in lowercase; otherwise if **S** is used the string is displayed in upper case. Note: The format can be used to output a value of any type. |
| **c** or **C** | character | **%c** outputs a character as it is, **%C** outputs a lowercase character in uppercase. |
| **n** |  | **%n** outputs a new line character appropriate to the platform running the application. You should always use **%n**, rather than **\n**. |
| **%** |  | **%%** outputs **%** |

**Note:** **Except for %% and %n, all format specifiers must match an argument; otherwise a run-time error: IllegalFormatConversionException is generated.**

**Width:**  The minimum total width of the formatted output, including a decimal point, a **+** or a **–** sign, if any.

**Precision**: For floating point (**double** or **float**), precision is the number of digits after the decimal point.

**Flag**: A character that specifies additional formatting:

|  |  |
| --- | --- |
| Flag | Description |
| '-' | The result will be left-justified. |
| '+' | A numeric output will always include a sign + or - |
| '0' | A numeric output will be zero-padded |
| ' ' | space will display a minus sign if numeric output is negative or a space if it is positive |
| ', ' | Groups of three digits in a numeric output will be separated by commas |

**Format Examples:**

Example1: The code:

**int i = 2;  
 double r = Math.sqrt(i);  
 System.out.format("The square root of %d is %7.3f%n", i, r);  
 double cost = 32.50;  
 System.out.printf("The cost is %.2f Saudi Riyals%n", cost);**

**System.out.printf("Number of people = %,d** **",5673458);**

produces the output:

The square root of 2 is 1.414  
 The cost is 32.50 Saudi Riyals

Number of people = 5,673,458

**Note:** Like the **print** method, the **printf** method does not generate new line. To generate new line **%n** or **\n** is used at the end of the format string.

# In the above example, the value 2 is output using the format specifier %d. The value of r is output using the specifier %7.3, it is output in a total width of 7 and with two leading blanks and 3 digits after the decimal point:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **1** | **.** | **4** | **1** | **4** |

The value of **cost** is output using the format specifier **%.2f** with two digits after the decimal point and in the minimum width required for 32.50:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 2 | . | 5 | 0 |

**Note:** Java raises a run-time error if the width of a format specifier is 0, for example: **%0.3f** is not valid

Example2: double num = -52.34578;

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | output | | | | | | | | |
| **System.out.printf("%-9.2f%n", num);** | **-** | **5** | **2** | **.** | **3** | **5** |  |  |  |
| **System.out.printf("%9.2f", num);** |  |  |  | **-** | **5** | **2** | **.** | **3** | **5** |

Example3: String city = “Dhahran”;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | output | | | | | | | | | | | | | |
| System.out.printf("%s%n", city); | D | h | a | h | r | a | n |  |  |  |  |  |  |  |
| System.out.printf("%S%n", city); | D | H | A | H | R | A | N |  |  |  |  |  |  |  |
| System.out.printf("%4S%n", city); | D | H | A | H | R | A | N |  |  |  |  |  |  |  |
| System.out.printf("%10s%n", city); |  |  |  | D | h | a | h | r | a | n |  |  |  |  |
| System.out.printf("%-10s%n", city); | D | h | a | h | r | a | n |  |  |  |  |  |  |  |

**Laboratory Tasks**

Task 01:

Write a complete Java program that declares and initializes the **volume** v of a sphere by a value in cubic centimeters. It then displays the volume in cubic centimetres and the **surface area** s of the sphere in square centimeters.

|  |  |
| --- | --- |
|  | π r3  s = 4 π r2 |

Sample program outputs:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Task 02:

Inflation is defined as the loss of purchasing power of a given currency over time. Let us assume that money loses 3% of its value every year. This means that an amount of money next year will equal to only 97% of its value this year.

Let us define the following equation to calculate inflation:

Write a complete Java program that declares and initializes a variable **currentAmount** to an amount in Saudi Riyals. It then calculates and displays:

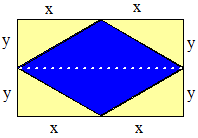
* the equivalent amount in a given year in the future.
* the equivalent amount in a given year in the past.

Sample Program outputs [assuming the current year is 2018]:

|  |  |
| --- | --- |
|  |  |

Task 03:

Consider the following figure:



Write a Java program that declares and initializes the lengths x and y. It then displays the lengths x and y, and finally it calculates and displays the area and the perimeter of the blue figure. Assume that the lengths x and y are in centimeters.

**Hint:** Area of triangle is (base \* height)/2

Sample program outputs:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Task 04:

Given force F in Newtons, acting between two electrically charged small spheres sphere1 and sphere2, the charge of sphere1 in Coulombs, and the distance between the charges in meters. Write a Java program that initializes the force F to 1.999438e-2 Newtons, the distance to 0.004 meters, and the charge of sphere1 to 5.25e-9 Coulombs. The program then finds and displays the charge of sphere2 in Coulombs. Use Coulomb’s Law:

F =

Where:

K = 8.9875 \* 109 Newtons Meter2 / Coulombs2 ; It is Coulomb’s constant.

**r** is the distance between the spheres in meters

**q1** and **q2** are charges of sphere1 and sphere2 in Coulombs.

**Note:** Define K as a constant and give it the value **8.9875e9**

**Note:** In Java a number can be written in Normalized Scientific notation by using the syntax:

**mantissaeIntegerExponent** or **mantissaEIntegerExponent**

**Example** 4.52e4 is 4.52 \* 104 and 3.567e-3 is 3.56 \* 10-3

The format specifier **%e** or %**E** is used in **System.out.printf** to output numbers in Scientific notation X.XXXXXXeXXX or X.XXXXXXe-XXX where the number of digits in the exponent is from 1 to 3

Output:



Task 05:

Write a Java program that initializes the amount of change, in Saudi Riyals, to be returned to a customer at a shop. The program then calculates and displays the minimum number of Saudi Riyal bills for the change and the equivalent number of 50, 10, 5, and 1 Riyal bills. Assume that the bills available at the shop are 1, 5, 10, and 50 Riyals.

**Hint:** Use remainder and integer division.

Sample program outputs:

|  |  |  |
| --- | --- | --- |
|  |  |  |