Java Programming Exercises - Hello World

Welcome! This guide will show you the basics of Java syntax and proper code style. Work through this guide by completing the tasks listed on their respective code files.

Let's Print!

Printing is how your Java program can send messages back to you. It's very helpful in letting you (or some other user) know what your code is doing, or for fixing errors/bugs (called debugging) in your software.

To print in Java, we use the following commands:

```
System.out.print("Hello World!");
System.out.print(3);
System.out.print(55.0);
```

> Note: See those semicolons at the end of each line? They're very important. They tell the Java compiler that this is the end of your statement, so make sure that you include a semicolon after each statement you write.

While these commands will print the desired text to your terminal, they will appear on the same line, which isn't desirable in most cases.

Output of above:

```
Hello World!355.0
```

We can instead use the following commands, which will insert a line break after each statement.

```
System.out.println("Hello World!");
System.out.println(3);
System.out.println(55.0);
```

Output of above:

```
Hello World!
3
55.0
```

It is best practice to use `System.out.println`, so that each print statement will appear on a new line. This makes your output more readable.

Task 1

Print your name.

Task 2

Print a message that indicates that your code is working on Task 2.

Comments

Comments are extremely important when documenting what your code does. A comment is a piece of text that is not considered part of the code. In Java, there are multiple ways to write a comment.

```
// This is a comment

[some random code] // Anything after the two slashes is ignored by the
Java compiler

/* I can also write comments that span multiple lines.
See? No need for slashes here!
Just remember to close the comment by using the pattern below..
*/
```

Comments are great for adding a blurb about what your code does, or temporarily preventing code from running, perhaps to diagnose a bug in your software. Adding comments to explain your code is good practice, especially if other people are going to read it...

Task 3

There's an error in the line of code in this section. Instead of deleting it though, comment it out, so it can be referred to in the future.

(Hint) To comment something out means to make that piece of code a comment. For example:

```
some code
System.out.println("Example");
some other code
```

To comment out the print statement, do the following:

```
some code
// System.out.println("Example");
some other code
```

Running your code

Let's take a quick segue and talk about how to actually run your Java code. There are two steps to running a Java program (enter these commands in your terminal in the same directory as the Java file):

1. Compiling

```
javac Main.java
```

2. Running

```
java Main
```

Some code editors have a feature where it can run your Java file for you. In that case, the above steps aren't necessary, but good to know anyways.

Try running your file. If you completed the above tasks correctly, you should encounter no errors, and see the messages that you wrote for Tasks 1 and 2 in your terminal window.

Variables and expressions

Variables are the core of programming. A variable stores information, such as a number, to be used later. Variables can also be modified, or destroyed (although destroying variables is a topic for a later date).

To create a variable in Java, we write:

```
[type] [name] = [value];
int myInt = 3;
String myString = "Hey!";
double pi = 3.14159265358979323846;
boolean iAmHavingFun = true;
```

> Look back at Task 3. Can you see why this line isn't valid code? Write a comment explaining why.

Note how we name variables:

Variables can be named whatever you want. However, there are some conventions that we follow:

> A convention is a rule that developers agree on; you don't have to follow them for your code to run, but other people who read your code will expect you to follow conventions for consistency and readability.

Variables are named using 'camel case'. This means that the first letter of each word is capitalized, except for the first letter of the first word (that's kept lowercase). Spaces are **not** allowed in variable names in Java (and pretty much all other languages).

```
iAmInCamelCase
IAmNotInCamelCase
i_am_also_not_in_camel_case
x // This is camel case
X // This is not camel case
```

Variable names should also be representative of what the variable is being used for. It's okay to have long variable names, rather than using `x` or `y` or `myVariable`.

```
int numberOfStudents = 15; // Good! I know what this variable is supposed to represent just from the name.
String xyz = "1600 Pennsylvania Avenue"; // Bad! Maybe a better name would be 'address' or 'whiteHouseAddress'
```

Task 4

Create a variable that contains your name. Make sure to name it accordingly.

> Hint: What type would your name be? Your name is a series of letters, so that would be a String.

Variables also act as placeholders. When you refer to a variable in your code, you are actually referring to whatever value that variable was assigned to. So, we can print variables:

```
System.out.println(numberOfStudents); // Prints 15
```

Task 5

Print your name, using the variable you created in Task 4.

We can also use variables to make more complex expressions, and set new variables to those expressions! For example:

```
int x = 5; // x == 5
int y = x + 3; // y == 8
int z = y - x; // z == 3
```

We also introduce comparison operators, which allow you to compare two expressions or variables or values. We use:

Operator	Meaning
==	equal to

Operator	Meaning
`!=`	not equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to

```
boolean b1 = y > z; // Is y bigger than z? Yes, so b1 == true.
boolean b2 = x >= y; // is x greater than or equal to y? No, so b2 ==
false.
boolean b3 = b1 == b2; // Is b1 equal to b2? No, because b1 == true, and
b2 == false, and true != false. So, b3 == false.
```

> Note: We use `=` to assign variables to the value of an expression. However, we use `==` to compare two expressions to see if they are equal to each other. So, you could write `int a = 10` and `a == 10`, but you **CANNOT** write `int a == 10` or `a = 10`.

While we can use `==` to compare numbers and booleans, we cannot use it to compare Strings (the reason why is out of the scope of this lesson, but will be covered later). Instead we can use:

```
boolean result = "hello".equals("bye"); // result == false
```

We can also reassign variables to a new value. That new value overwrites the old one. When reassigning a variable, the format is as follows:

```
[previously made variable] = [new value];
```

For example:

```
int x = 5; // x == 5

int y = x - 3; // y == 2
System.out.println(y); // Prints 2

x = 6; // x == 6
System.out.println(y); // Prints 2 (y was never recalculated with the new
value of x)

y = x - 3; // y == 3
System.out.println(y); // Prints 3
```

```
y = x + y; // y == 9
// Note: in the above example, the reference to 'y' to the right of the
'=' is the value of 'y' before running that line. 'y' is then set to be
the value of this expression. In summary, the expression is evaluated
first, then the variable is assigned/reassigned.
```

Task 6

Next to each line, write the value of what you think the expression will be in a comment. Check your work by printing the variables and running the code. The first three are done already, as an example.

Conditional Logic

We want our code to be able to 'think' on it's own, to be able to do different things based on different states. One way we can do that is through an `if` statement. An `if` statement allows you to perform a specific block of code **only** if an expression evaulates to `true`. We write `if` statements as follows:

```
if([expression]) {
    [code];
}
```

If [expression] is `true`, then [code] will be run. Otherwise, it will be ignored. For example:

```
int x = 10;
if(x > 5) {
    System.out.println("x is bigger than 5"); // This code will run
}
if(x - 8 < 2) {
    System.out.println("x is less than 10"); // This code will NOT run,
because x - 8 == 2, and 2 is NOT less than 2 (2 is equal to 2)
}
```

The expression inside the if statement must evaluate to a boolean. So, `if("hello") ...` is invalid code.

We can also program an action to occur if the expression in the if statement returns false. We do so as follows, using the `else` clause:

```
if(x < 5) {
    System.out.println("x is less than 5");
} else {
    System.out.println("x is greater than or equal to 5");
}
```

We can also check multiple expressions in one statement using an *if-else ladder*.

```
if(x < 5) {
    System.out.println("x is less than 5");
} else if(x < 10) {
    System.out.println("x is greater than or equal to 5, and less than
10"); // We know that x >= 5, because the first expression in the if-else
ladder (x < 5) failed. When an if-else ladder is run, each 'if' expression
is checked in order from top to bottom.
} else if(x < 15) {
    System.out.println("x is greater than or equal to 10, and less than
15"); // Same logic as above
} else {
    System.out.println("x is greater than 15"); // If the code in the
'else' block is run, we know that all of the expressions in the if clauses
above are false.
}
```

Task 7

Create an if-else ladder that does the following:

1. If `greeting` is "Hello", print: `"Java said hello!"`
2. If `greeting` is "Bye", print: `"Bye Java!"`
3. If `greeting` is none of those, print: `"Invalid greeting"`

> Hint: Remember that we can't use `==` when comparing Strings. We must use the `.equals()` method (we'll talk about what a method is in a future lesson), so for example: `greeting.equals("Howdy")` would compare the value of the variable `greeting` to the String `"Howdy"`.

Task 8

Write an `if` statement that prints a message if `greeting` is longer than 8 characters (a character is a letter/number/symbol). You can check the length of a string by using the `length()` method, like so:

```
int x = "Hello World!".length(); // x == 12
String myString = "Goodbye Moon...";
int y = myString.length(); // y == 15
```

Loops

Oftentimes, we run the same task in our code multiple times. Instead of writing the same code out several times and cluttering our code, we can use a loop to run whatever we want a certain number of times. Loops are also useful later when we talk about data structures, and when we may not know how many times to run a piece of code until we have to run it.

There are two types of loops that we will talk about today (there is a third, but that is a topic for a later time).

`for` loops

A `for` loop uses an `int` to keep count, and a start/end value to define how many times to run a piece of code. For example, we can write:

```
for([Declare a new variable or use an existing one]; [An expression that must be true for the loop to run]; [A statement that increments the variable being used in the loop (otherwise, the loop would run forever)])
{
    [code]
}

for(int i = 0; i < 10; i++) { // We created a new variable called i and set it to 0. Then, we made the loop ensure that i must be less than 10 for the loop to run. Last, we set that every time that the loop runs, we will increment i by 1 (see note that follows).
    System.out.println(i);
}
```

> Note: We can write `+=` as a shorthand to increment an `int` by a certain amount and assign that int to the new value...all in one step. So `i += 1` is the same as writing `i = i + 1`. Furthermore, `++` is just shorthand for `+= 1`, since it is used so often.

The above code would produce the following output:

```
0
1
2
3
4
5
6
7
8
9
```

There are several things to note here:

1. The loop runs 10 times, as noted by the fact that 10 lines are printed (that means that there were 10 calls to `System.out.println`, and therefore, we can conclude that the loop must have run 10 times).
2. The first number printed is `0`, not `1`, and the last number printed is `9`, not `10`. The number being printed is called the **index**, since it represents the state of the loop. A common notion in computer science is called *zero-indexing*, which basically means that we start counting from 0 (this applies to loops, such as in this case, but can also apply to other things, such as Arrays, which we will investigate later). While you could start the loop at 1, it is common convention to start at 0.

Some more examples of `for` loops:

```
int x = 1; // We can use an existing variable in our for loop as well.
for(x; x <= 10; x += 2) {
    System.out.println(x * x);
}
```

Outputs:

```
1
9
25
49
81
```

> Try to understand why this loop prints out what it does before moving on.

```
for(int lineNumber = 0; lineNumber < 10; lineNumber++) {
    for(int spaceNumber = 0; spaceNumber < lineNumber; spaceNumber++) {
        System.out.print(".");
    }
    System.out.println("*");
}
```

Outputs:

```
*
.*
..*
...*
....*
.....*
......*
.......*
........*
.........*
```

> Try to understand why this loop prints out what it does before moving on.

A brief detour to talk about scope:

Let's talk about scope. Scope is the area of your code in which a variable is defined. When you create a variable in Java, it is not accessible from everywhere (this is mostly for organization and security...imagine if you had to use a different letter for each for loop you make!)

Generally, if I create a variable inside some set of curly braces ({}), I can only access it from inside those curly braces. Outside, that variable didn't exist (Variables created inside the header of a for loop count as being

inside the curly braces). For example:

```
int x = 10;
for(int y = 0; y < 5; y++) {
    int z = 3;
    x += z + y;
    // Inside the for loop, I have access to x, y, and z.
}
// However, outside the for loop, y and z are not defined. I can still
access x though, and it's value will no longer be 10, since the for loop
modified it.

if(x < 5) {
    String str = "Can you read me?";
    // Inside the body of the if statement, I can read the value of str
(and print it or whatever I want).
}
// However, outside, str doesn't exist. If I try to access it, I will get
an error.

// Instead, I can do the following:
String str = ""; // The empty string
if(x < 5) {
    str = "Can you read me?";
}
// Now, because str was created outside of the if statement, I can read
it, because I am still in its scope.
```

`while` loops

While loops are similar to `for` loops, but they only check if an expression is true or not. I can write:

```
while([some expression]) {
    [code]
}

int counter = 0;
while(counter < 10) {
    System.out.println(counter);
    counter++;
}
```

Outputs:

```
0
1
2
3
```

```
4
5
6
7
8
9
```

More examples:

> Note: make sure you understand how these loops work before proceeding

```java
int myNumber = 0;
while(myNumber < 50) {
    myNumber += 7;
}
System.out.println(myNumber); // Output: 56
// What is the significance of myNumber?
// Answer: It is the first multiple of 7 bigger than 50. How did the loop
get us this result?
```

> In the vast majority of cases, `for` loops can be written as `while` loops, and `while` loops can be written as `for` loops. So, when programming, choose what will be the simplest to write for the situation at hand.

Task 9

Write a `for` loop that prints out the first 10 even numbers.

Task 10

Complete the above task, but using a `while` loop instead.

Task 11

Print the sum of the first 10 positive integers using a `for` loop (Compute 1+2+...+9+10, and print the answer).

Task 12

Complete the above task, but using a `while` loop instead.

Task 13

Print the sum of the first 10 perfect squares using a `for` loop (Compute 1^2+2^2+...+9^2+10^2, and print the answer).

> To compute a perfect square in Java, just multiply the number by itself for now (so 5^2 -> `5*5`), there is a better way, but let's save it for a future lesson.

Task 14

Complete the above task, but using a `while` loop instead.

Task 15 *(bonus)*

Using a loop (either a `for` or `while` loop), find the number of perfect squares whose sum is the closest to 500, while still being less than 500 (Find the largest `n` such that 1^2+2^2+...+(n-1)^2+n^2 < 500)