A Beginner’s Guide to Gelli

So, you’ve decided to learn some programming and you Gelli is your language of choice. Well, you’ve come to the right place. Here we will go over the basics of programming concepts while learning Gelli. If you already have experience programming and are just looking for syntax references, feel free to jump around, but for those of you who are new to this, I recommend following this guide straight through. We will be building on concepts as we go.

# Table of Contents

1. [Arithmetic](#_Arithmetic)
2. [Comparison Operators](#_Comparison_Operators)
   1. [Boolean Symbols](#_Boolean_Symbols)
   2. [Boolean Words](#_Boolean_Words)
3. [Assignment Operator](#_Assignment_Operator)
4. [Grouping Symbols](#_Grouping_Symbols)
   1. [Parentheses](#_Parentheses)
   2. [Arrays](#_Arrays)
5. [Program Body](#_Program_Body)
   1. [Program Section](#_Program_Section)
   2. [Definitions Section](#_Definitions_Section)

5.2.1. [Structures](#_Structures)

5.2.2. [Functions](#_Functions)

5.2.3. [Bringing it all Together](#_Bringing_it_all)

1. [If-Else Statements](#_If-Else_Statements)
2. [Loops](#_Loops)
   1. [Repeat-Times](#_Repeat-Times)
   2. [Repeat-Forever](#_Repeat-Forever)
   3. [Repeat-For](#_Repeat-For)
   4. [While](#_While)
3. [Links](#_Links)

# Arithmetic

If you’re seeking out programming, then you should be familiar with basic math concepts. Gelli, like other programming languages, supports addition, subtraction, multiplication, division, and exponents. The usage of such operators is straightforward. Let’s use the common Facebook problem as an example:

6 / 2 \* (1 + 2)

Every operation must be explicitly defined in Gelli, so dropping the multiplication symbol isn’t an option, but I digress. Gelli will analyze this statement and execute it from left to right using the order of operations. First, it will do 6 / 2. Here, the ‘/’ takes place of , a common practice in programming languages since that symbol is very difficult to type and the slash is representative of fractions for most math people anyway. This comes out to be 3. Next, it sees the set of parentheses and knows they need to be simplified before continuing. It will add 1 and 2 to get 3 inside the parenthesis. Lastly, we are left with 3 \* 3. Once again, in programming languages you will often see that \* takes the place of • or x for multiplication. We don’t use x for the same reason we don’t use it in algebra, we use variables like that and don’t want to be confused. We don’t use • because it is too difficult to type. Thus, using \* has been our decision. It kind of resembles the dot and x anyway, doesn’t it? Anyways, 3 \* 3 is 9 and we have our answer!

## Mod

Sometimes in programming we want to do division, but we don’t want the full answer, we just want the remainder you would get after division instead of full value. For example, when you divide 5 by 2 you would normally get 2.5. However, the whole number part is 2 and when doing division by hand you would have 1 left over before moving onto decimal places. This 1 is the number you would get if you used mod instead of /.

10 mod 3

23 mod 7

6 mod 7

By normal division, you would get 3.333333, 3.285714, and 0.857142 respectively. However, mod takes those decimals and puts them into whole numbers like what would be in the numerator of a mixed number. 10 mod 3’s mixed number is , so mod returns 1. 23 mod 7’s mixed number is , so mod returns 2. 6 mod 7’s fraction is , so mod returns 6.

# Comparison Operators

## Boolean Symbols

Remember those comparison operators? You don’t hardly use them much after you learn them aside from graphing, but in programming they become significantly more useful. The less than: ‘’, less than or equal to: ‘’, greater than: ‘’, greater than or equal to: ‘’, not equal to: ‘’, and equal to: ‘’ comparisons are something we like to call “boolean operators”. Languages use these to determine when we should do things and when we shouldn’t. Languages use them around if, while, and for code, but we’ll cover those later. For now, let’s understand how these work.

First off, I should preface by saying that how I represented these above is not entirely accurate to how they are represented in programming languages. You’ll notice that , , and are on your keyboard, but the rest are not. Thankfully, the way we represent these are straightforward. and are represented by <= and >= respectively, and is represented by !=. You will find in your programming career that exclamation points are used in boolean comparisons to negate. If it would be true, it becomes false. If would be false, it becomes true. Not too hard. Now, let’s get into a few examples:

5 + 2 < 8

3 / 4 > 1

In the first example, 5 + 2 is 7 which is less than 8 so it is true. In the second one, 3 / 4 is 0.75 which is less than 1, so it is false. Let’s look at another set that shows how the less than or equal to operator works.

6 + 1 <= 8

6 + 2 <= 8

6 + 3 <= 8

Here, the first line breaks down to 7 <= 8 which is true. The next line breaks down to 8 <= 8 which is also true. However, the last line breaks down to 9 <= 8 which is false. The greater than or equal to operator works in the same way. Lastly, let’s look at the equal to and not equal to operators.

9 != 8

7 != 8

8 = 8

All of these are true, but let’s briefly explore why. 9 and 7 are clearly not 8, so they are not equal to, or !=. 8 is 8, so it equals 8. Easy!

## Boolean Words

There are other ways to get some true and false values. and and or are words that allow you to combine multiple boolean expressions together. I’ll try to keep this simple. If the operations on both sides of and are true, then it is true. Otherwise, it is false. If at least one of the operations on either side of or are true, then it is true. Otherwise, it is false. Let me show you what I mean:

5 >= 4 and 5 >= 2

5 < 2 or 5 > 2

5 <= 4 and 5 >= 2

5 < 2 or 5 < 3

In the first set, both statements are true. With the and, 5 is greater than or equal to 4 and 5 is greater than or equal to 2. Since both are true, the whole statement is true. With the or, 5 isn’t less than 2 but is greater than 2 so the statement is true. In the second set with the and, 5 is not less than or equal to 4, so the whole statement is false because it was an and. With the or, 5 is not less than 2 and 5 is not less than 3, so the whole statement is false.

If that’s confusing and you can’t seem to figure it out, there are plenty of online guides that explain this concept. Just lookup “logical and or” and pick your favorite website or video.

# Assignment Operator

Earlier I mentioned that we can use x like you can in algebra. In fact, you can use almost any word you want! If I want to remember a particular number, I can name it in such a way that I’ll remember. We call these variables because their value can vary. To put a value into a variable, we use the equals sign again: =. Gelli is smart and will look at the situation you are using = in: if you are assigning a variable, it will use it like that. If you are checking if two things are equal, it will use it like that. Let’s look at how Gelli uses this operator:

answer = 4 + 2

pi = answer – 3 + 0.14

Now, any time we use the word answer, Gelli will use 6. Anytime we use pi, Gelli will use 3.14. This is all fine and dandy, but in truth, we missed something. See, there are other things you assign to variables in programming languages besides numbers. Other common things that get variable names are text and arrays. We have to tell Gelli the first time we use a variable what kind of variable it is. So, the above should look like this:

answer: number = 4 + 2

pi: number = answer - 3 + 0.14

I should also note that there is another type besides numbers. There are also texts. They will be useful later but for now just know this is what they look like and that they can be added together. You cannot do any other operations on them besides addition.

hi: text = "Hello"

welcome: text = hi + " World!"

Whenever you create a variable name, you can’t just put anything in. Any words that are used by Gelli to recognize words cannot be a name. You can put numbers in the name, but it can’t start with one. To be clearer, you can use letters, numbers, and underscores ‘\_’ in your variable name, it simply cannot start with a number or be a word Gelli already uses. Here are a few examples of acceptable variable names:

car

number1

shoesize

Computer\_number

\_123Gel\_

The first word seems simple enough, but the second word has the word number in it. However, since we put 1 at the end, Gelli does not recognize it as the key word ‘number’ that we used to tell Gelli that our variable is a number, but as ‘number1’. shoesize checks out fine, and so does Computer\_number, our first example of an underscore. The last one starts out with an underscore and then has numbers and letters and ends with an underscore. I would not recommend creating variable names like this, but it is technically allowed.

Something to note is that Gelli is not case sensitive to variables or key words. In many other programming languages, it is very important to make sure that everything uses capital letters consistently, but Gelli does not care. A letter is a letter regardless of whether it is lowercase or uppercase.

# Grouping Symbols

## Parentheses

We already used parentheses in our [Arithmetic Operators](#_Arithmetic) section. They tell Gelli that that section should evaluated, or completed, first before using it with the code around it. You can nest these parentheses to be more specific. Here’s an example:

2 \* (6 / (4 + 3))

If we didn’t have parentheses, you might say the answer to this is 6. If we couldn’t nest parenthesis, you might say the answer to this is 9. However, because we can put them together the real answer is 1.7142857142857142. The order things are done in can be super important, so we use parentheses to make sure things happen the way we want them to.

## Arrays

Next, let’s look at square brackets. These show the beginning and end of arrays. Since we’re on the subject, I’ll show you how to assign one of these to a variable too. Let’s take a look:

x: array[10] of number = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

That’s a lot all at once, let’s break it down a little. First, we declare the variable name x. Easy enough. Next, we use the word array. This tells Gelli that we are declaring an array of items. Arrays are just a list of things in order. Not too difficult. Then we have [10]; this tells Gelli that we want our array to have 10 things. We use the word of to indicate to Gelli that the next word is a key word with the type of things that will be in our array, we are using numbers for now. The = is the assignment operator we talked about earlier. Lastly, we get to the actual array. We start and end with the square brackets: [ ]. Then we just fill the brackets with the values we want, in this case numbers, separated by commas. Make sense? Good, let’s continue.

Something about arrays is that they can have multiple dimensions. Now, it’s not as confusing as it sounds. Let’s look at a 2D array first:

x: array[2,3] of number = [[3, 1, 4], [1, 5, 9]]

x: array[2,3] of number = [[3, 1, 4],

[1, 5, 9]]

These two are the same thing but written in a way that shows how Gelli sees it and how we might see it. Let’s look at the differences between these two and the first array declaration. The first difference is the numbers next to array. There are 2 in there. This tells Gelli that there are 2 dimensions, and the sizes for each are listed. To break it down a little more, there are arrays of 3 items, and there are 2 of those arrays. If you are familiar with matrices, you should be right at home. If not, don’t worry about it, this should be simple enough to grasp. The first array has 3, 1, and 4 in it. The second has 1, 5, and 9 in it. Each additional dimension works the same way, we just put another array of the specified number of elements in the previous array.

You might be wondering, “how do I even use this?” Well, we use indexing to reference what number should be pulled out. Let’s take a look:

x[2]

x[1,3]

The first line would get the whole array [1, 5, 9]. The second would get the third item from the first array which is 4. Not hard at all!

Something else to know is that you can specify what range of numbers you want to index by. For example, if you want to count your array from -4 to 5, you can say:

x: array[-4 to 5] of number = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

This shows Gelli that you want the first number to be -4, and the last number to be 5. If we asked for x[1], we would get 9. You can also do this in arrays with multiple dimensions too.

x: array[0 to 1,3] of number = [[3, 1, 4], [1, 5, 9]]

In this example, the first array is referred to as 0 while the second is 1.

# Program Body

## Program Section

The code you write in its entirety is called a program. Programs in Gelli have two parts: definitions and program. Let’s look at the program first.

The start of anything you do begins after the word program. Programs are started with program and ended with end program. Let’s take a closer look:

1. program
2. x: number = 4 + 5
3. end program

You may have noticed there are line numbers in this example. Programmers use line numbers to identify where particular pieces of code are at more quickly. They are not something that you will type when writing your programs and they do not affect how your code runs. Next you will notice that line 2 is indented. This is not required either, however, with large amounts of code it helps keep track of how the code is structured. The most common indenting amounts are two and four spaces. I prefer four, so it’s what I used. This program adds 4 and 5 and assigns the result to x, then ends.

## Definitions Section

Next, we have the definitions section. It is not entirely necessary to have, but can make your code much easier to write. The definitions section has 3 parts that are all also optional, but if any 2 or more are combined, must be put in this order: structures, globals, and functions. The global section just defines variables that you want to be available everywhere in your code, but structures and functions require more explanation. I will explain them more in depth later, but I want to show you an example of them now. I do not expect you to understand everything now, so remain calm and don’t freak out on me when you see this next part. Just, look at it, wave hello, and we’ll talk about it.

1. definitions
2. structure point
3. x: number
4. y: number
5. end structure
6. pi: number = 3.1415926
7. function add\_points(p1: point, p2: point) returns point
8. p3: point = {p1.x + p2.x, p1.y + p2.y}
9. return p3
10. end function
11. end definitions
12. program
13. point1: point = {4, 5}
14. point2: point = {1, 2}
15. point3: point = add\_points(point1, point2)
16. end program

I know, I know. I threw a lot at you at once. Let’s take a deep breath and look at break it down. Look at line 15. There’s the familiar program section, but it has some funny looking variables in it and words that have parentheses next to them. To understand what these things do, we have to look at the definitions section above. Like the program section, the definitions section begins with definitions and ends with end definitions. Remember that this section can have up to 3 sections in it: structures, globals, and functions.

### Structures

The first part is structures, so let’s grab that part and look at it a little more closely.

1. structure point
2. x: number
3. y: number
4. end structure

The definitions section is all about defining what things mean, and that’s exactly what we’re doing here. We are telling Gelli that there will be a new type of structure called a point. But what even is a structure? There are two ways of looking at structures. You can either look at them like arrays that use variable names instead of numbers, or you can look at them like representations of more abstract concepts. Although the first way might be okay while you’re learning, I really recommend the latter.

So, we have this structure, and it has 2 lines inside of it. These are variables that the structure will hold onto. They look like assignments for variables we did earlier, but without the equals sign. That is important to recognize because that’s what these are, but we cannot assign values to these variables in the definition of a structure. Why? Because we are generically telling Gelli what one of these looks like. We could have multiple structures, and each one can have different values for each variable.

So, this point structure has two variables, x and y, that are both numbers. I designed this structure with graphing points in mind. Like we do in algebra. It gives the x coordinate and y coordinate for the point. You can also put in text and other structures, but for now we just have two numbers. But how do we use them?

p: point = {2, 3}

This feels reminiscent of arrays, but there are a few key differences. First, structures use curly braces instead of square brackets. Next, since this isn’t an array, we don’t have to tell Gelli how big the structure is because we already defined it in the structures section. When we are making a new instance of structure, or in this example, when we are making a new point, we have to remember what order we declared the variables in. We made the point by declaring x then y, so that is the order they get assigned in. x gets 2 and y gets 3. Now we have to figure out how to get those values back out. Here’s an example.

x: number = p.x + p.y

In this example, x will get the value of x and y added together which is 5. We accomplished this by referencing the point we made and called p, putting a period, and then the name of the variable within the structure. This is a common practice in programming. Periods often show ownership of something to something higher up. The structure p owns the x and y variables, and we want p’s versions of them. If we declared another point structure somewhere, we wouldn’t want to get both sets or override one with the other, so this helps us keep track of who owns what.

### Functions

functions are an important part of modern programming. It allows us to reuse code and make it more understandable. You first tell Gelli you are making a function, then the name of the function, the arguments, and what it returns. “What are arguments? What does return mean?” I hear you ask through the page. Well, arguments are variables we give to functions for them to use in their miniature programs. Then functions return a value. This means that they have computed something for us and want to give us that value to use in our main code. Let’s jump down to line 9 and take a look at a function in action:

1. function add\_points(p1: point, p2: point) returns point
2. p3: point = {p1.x + p2.x, p1.y + p2.y}
3. return p3
4. end function

This function is called add\_points. When we ‘call’ (the verb we programmers use when we are using a function) add\_points, it requires 2 arguments. Remember that variable definition stuff we were doing? Well, here it is again: the variable names are p1 and p2, and both variables are point structures. Lastly, this function returns a point structure. So, what does it do to create this new point? Look at the code between function and end function. The function creates a point structure called p3. We decided to add the points’ x’s and y’s together to create a new point. Not particularly useful to my knowledge, but it serves it’s purpose in explaining my point. We used the arguments to the function to help create something new. Then we gave that something new back to the main part of the program by using the return key word. Make sure that you remember to use returns when telling Gelli what the function will return on the first line and return without the s when telling Gelli what the value should be after computing.

Something else you can do with functions is specify something as changeable. What this does is allow you to modify an argument’s value back in the original place of the function call. Here is a short example:

1. definitions
2. function add\_five(x: changeable number) returns nothing
3. x = x + 5
4. end function
5. end definitions
6. program
7. num: number = 5
8. add\_five(num)
9. end program

Here the function add\_five takes a number and changes it’s value by adding 5. After the function ends, num’s value will be 10 instead of 5.

### Bringing it all Together

Let’s take a look at the whole thing all together again. Ready?

1. definitions
2. structure point
3. x: number
4. y: number
5. end structure
6. pi: number = 3.1415926
7. function add\_points(p1: point, p2: point) returns point
8. p3: point = {p1.x + p2.x, p1.y + p2.y}
9. return p3
10. end function
11. end definitions
12. program
13. point1: point = {4, 5}
14. point2: point = {1, 2}
15. point3: point = add\_points(point1, point2)
16. end program

This time we were ready for it. This program creates point structures, a global variable called pi, a function called add\_points, and runs by creating two points and calculating a third with add\_points. Yay! We did it! Not so bad after all. I’d also like to take this opportunity to point out some of my spacing and indentation choices. While not necessary for the code to run smoothly, it sure makes everything more readable. Spacing helps readers understand different sections of code. It’s kinda like a paragraph in writing. Similar ideas should be grouped together. The indentations also help with this concept of paragraphs where everything that is indented to a certain point goes together with the less indented item above it.

Also, notice how we used the function on the other end of an assignment. We used it as if it was already the value we wanted. add\_points essentially replaces itself with the value it returns, so the assignment for point3 actually sees this:

point3: point = {5, 7}

Now you may have noticed we didn’t actually use pi. Normally if you don’t use something, it’s best to remove it from your program, but I’m using it as an example. This variable is different than ones declared in the program section. We can use it anywhere, even in functions that weren’t given it as an argument.

1. definitions
2. pi: number = 3.1415926
3. function print\_pi() returns nothing
4. print(pi)
5. end function
6. end definitions
7. program
8. print\_pi()
9. end program

Here’s something to note. Most instances of Gelli have some pre-defined functions. print is one of those. What does print do? print is a word we programmers have grown accustomed to when wanting to display our information to a terminal. Just think about running the program and then it actually telling you what the result is. Previously, all the programs we have created would run silently and tell us nothing after it finished. By using print, we can actually find out the result of what we ran. This is where text comes in handy. Anything we print eventually gets converted to a text before being displayed.

But I digress, I wouldn’t recommend writing a function that does what I did above since you could do that in the program section just as easily, but it allows me to show off that we didn’t have to give the print\_pi function any arguments for it to be able to access the pi variable. Notice how even though we didn’t have any arguments, we still needed the parentheses. Also, even though our function doesn’t return anything, we still have to tell it *something*. So, Gelli has the nothing key word for when the things that are computed do not matter after the function is over. You will use this from time to time, especially when after you calculate a value you print it out in the function.

# If-Else Statements

Next let’s look at if and else statements. These are cornerstones in programming as they allow us to make decisions on what to do based on the values of certain variables. They use the comparison operators we looked at earlier in section [2. Comparison Operators](#_Comparison_Operators). When something is false we don’t move forward with the code inside, but we do when it is true. Here’s your first look at some if statements:

1. program
2. x: number = 5
3. if x > 10 then
4. print("x is greater than 10")
5. end if
6. if x <= 10 then
7. print("x is less than or equal to 10")
8. end if
9. end program

After we assign 5 to x, we check if it is greater than 10. If it is, we’ll print that text. Then if x is less than or equal to 10 then we’ll print the other text. This is okay, but if you think about it, if x is not greater than 10, it must be less than or equal to 10, so there’s a way we can clean this up a little.

1. program
2. x: number = 5
3. if x > 10 then
4. print("x is greater than 10")
5. else
6. print("x is less than or equal to 10")
7. end if
8. end program

I know we didn’t save much room here, but we certainly made the program more readable. In that case, x can only be one or the other so we can use if and else together to print what we want. Both programs will display x is less than or equal to 10. What if we want to add a third statement that prints when x is equal to 10? Well, let’s take a look:

1. program
2. x: number = 5
3. if x > 10 then
4. print("x is greater than 10")
5. else if x < 10 then
6. print("x is less than 10")
7. else
8. print("x is equal to 10")
9. end if
10. end program

Here we combined the key words else and if together to chain a series of conditions together. To dumb it down, if the first thing is true, do the first print; otherwise, if the second thing is true, do the second print; otherwise do the last print. It’s worth noting that when executing the code Gelli and other languages will skip the rest of the statements once it matches with one. So, in this instance, Gelli doesn’t consider the code from the else statement because it matches with the else if statement and displays x is less than 10. You can put as many else ifs back-to-back as you want, and they don’t have to end with an else. You can have if statements without else ifs or elses, and if elses without elses. However, to have an else if or an else, you must first have an if.

# Loops

Now let’s get into some more reusable code concepts. We already learned about functions, but there is something much more basic we can use if we want to run the same section of code multiple times over in one spot. Loops are an essential part of programming and there are many types. In Gelli we have repeat-times, repeat-for, repeat-forever, and while. Many use the [Comparison Operators](#_Comparison_Operators) we talked about before as well. Let’s dive in!

## Repeat-Times

1. program
2. x: number = 0
3. repeat 5 times
4. x = x + 1
5. end repeat
6. print(x)
7. end program

repeat < number > times is a very important operation in programming. If you want to run code a set number of times, this is your man. This loop will execute as many times as you tell it. Here, we have it add 1 to the value of x 5 times. This results in the print statement displaying 5.

## Repeat-Forever

1. program
2. x: number = 0
3. repeat forever
4. if x > 10 then
5. break
6. end if
7. x = x + 1
8. end repeat
9. print(x)
10. end program

repeat forever is like repeat times, but you put in infinity. This will run until your computer stops the program or your computer stops working altogether. However, there is a way to stop it. All loops take a couple of key words that mess with the loop a little. The first one is break. This will break out of whatever loop you are in. In this instance, we are in a repeat forever loop, but if x becomes greater than 10 we want out. The print statement will display 11 at the end as a result.

## Repeat-For

1. program
2. arr: array[5] of number = [1, 3, 5, 7, 9]
3. repeat for all num in arr
4. if num = 5 then
5. continue
6. end if
7. print(num)
8. end repeat
9. end program

repeat for lets you use all the elements in an array in one loop. See we declared an array of 5 numbers. Then we call a repeat for loop. This runs the loop for every number in the array arr. Notice I put for all num in arr? num is not a variable we have declared. However, it is a variable I am declaring for the loop to represent each number in arr. In other words, num will be 1, 3, 5, 7, and 9 depending on the loop number we are on. It goes in order from left to right, so no worries there. However, this program does not simply print out all five numbers. If num is 5 we continue. What does continue mean? Well, it’s another one of those loop words we can use like break. It means skip the rest of this time and start the next one. In other words, the program will print 1 and 3, skip 5, and then print 7 and 9. Neat!

## While

1. program
2. x: number = 10
3. while x > 0
4. x = x - 2
5. end while
6. print(x)
7. end program

The last loop. while loops run while the condition is true. That means that you’ll need to update at least one of the values in the condition inside of the while loop, otherwise you’ll be stuck here forever! In this program, I set x equal to 10 and subtracted 2 from it until it becomes less than 0. The print statement will display -2 after it’s all said and done. Now, it’s worth noting that repeat forever and while can be used very similarly. I would recommend trying to use a while loop unless your repeating condition is just that convoluted. Many other languages do not have a repeat forever statement, but it can be easily be accomplished by ensuring the condition will never be false. Now, if you had a program that should just run until you stop it which to the program is forever, then maybe repeat forever is for you.

# Links

We’re coming up on the end of things to learn about Gelli, but I still have one final thing to teach you about: links. Links relate to one of the harder things for programmers to learn. Many programmers struggle with these concepts when they begin to learn C, another programming language. But fear not! Gelli has been written in a way to simplify the concept and but keep its usefulness at the same time; let’s dive in!