A programmers thoughts on R: Is it better than traditional languages?

Harley Velez

Is R superior to a more traditional programming language like Java? Maybe, maybe not; in this blog post we'll take a dive into both worlds and find out for ourselves. Regardless of the field you work in, if you have a task that requires computation you'll need to ask yourself what the best language to use is. I'll be using different Java syntax throughout this post, but don't fret if you haven't seen Java before as I'll explain everything you need to know. We'll compare them in several ways, datatypes; OOP; data structures; and ease of use.

> "R is a language and environment for statistical computing and graphics … R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, …) and graphical techniques, and is highly extensible." - r-project.org

Lets first learn the basic data types of each language,

Data Types

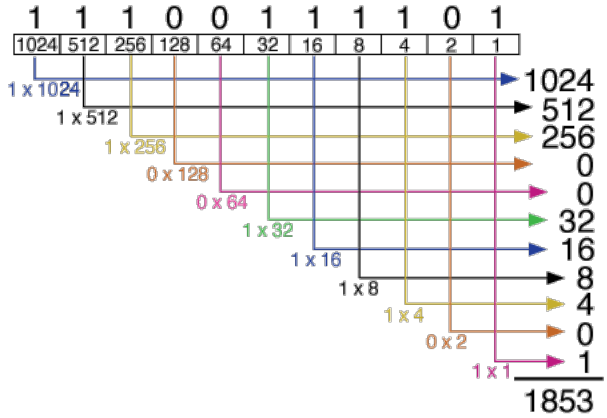
R has several data types (for more info see this tutorial:

- Integer (32-bit)
- Double (64-bit)
- Logical
- Complex (Rarely Used)
- Character (String)
- Raw (Rarely Used)

Java data types (for more info see the official Java documentation):

- byte (Binary representations of numbers, 8 bits in one byte)
- short (16-bit integer)
- int (32-bit integer)
- long (64-bit integer)
- float (32-bit double)
- double (64-bit double)
- char (A single string character)
- String (A sequence of string characters)

A note for those new to bit representations. As you may have heard before computers store everything as 1's and 0's. A number is represented by these ones and zeroes. The simplest way to read it is base 2 (humans count in base 10). The image below gives an example of this form of counting,



We can imagine each slot from right to left as being powers of 2. So the first box on the right is 2^0, next 2^1, 2^2, and so on. The 1's and 0's mean either the box is ticked on or off. Any box with a 1 means to use this power of 2. It turns out that any number can be represented by powers of 2. An important connection is that a number (that's not a power of 2) can be represented by combinations of powers of 2 smaller than it. For example, 7 can be represented by 2^2 + 2^1 + 2^0. In binary this is 0111.

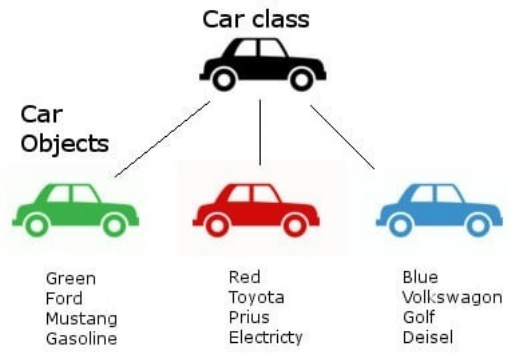
Modern day computers actually use more sophisticated ways of storing numbers in order to save negatives and decimals but everything rests on the basic ideas above. The core takeaway if all the powers didn't make sense is that the bigger the number the more bits we need to represent it, so for instance sometimes a user might want to choose a short since they know they'll only use up integers less than 2^16. If you didn't understand and want to or you understood and want to learn more check out this beginners guide on binary.

On one hand we can say that the amount of different datatypes Java provides allows for more flexibility and also could help to store things more efficiently when working with big data. But on the other hand we could view this as more complexity for the user. But, this round goes to Java not just for the flexibility but also for the clarity, as R has blurred the lines for their users by also using what are called "data modes". These modes represent numeric, logical, etc. They are put ontop of the existing data types to categorize them. So in my eyes, the level of simplicity gained from having a few data types is obscured by a hierarchal structure.

Objects

Both Java and R are languages that force Object-Oriented Programming (often referred to as OOP) onto the user. What is OOP? It can be thought of as an abstraction or an encapsulation technique. An object can be thought of as a container that holds certain data types or even other objects. Since everything in R is considered an object an object container in R most in turn hold other objects.

We'll call our container a class. Say we wanted to create a car class (following the example started on this quora post). As follows,



In Java this would look something like,

```
public class Car {
  # Internal Variables

    String color;
    String Make;
    String Model;
    int gas;

  # Constructor function that takes in attributes and assigns them to
  # internal variables when we initially make a new car

    public Car(String color, String Make, String Model, int gas) {
    this.color = color;
    this.Make = Make;
    this.Model = Model;
    this.gas = gas;
    }

  # Sample function to fill gas up

    public void fillGas(int amount) {
        this.gas = this.gas + amount;
    }

}
```

In this example we can ignore the “public” tags. We can see there’s a class defined as Car that holds four attributes, color, Make, Model, and gas. There is a function in this class that is called whenever a new car is made using this class that takes in those four arguments and saves them to the internally defined variables using the keyword this.

Now in our code we can create cars and access its attributes as follows,

```
# Create a car object and store it in the variable exampleCar
Car exampleCar = new Car(“Red”, “Lamborghini”, “Gallardo”, 5);

# Access the color attribute
exampleCar.red;

# Call the fillGas function
exampleCar.fillGas(4);
```

A note and exercise for the reader: For simplicity we broke a few rules of proper OOP programming. In proper OOP we wouldn’t want to directly access a classes variables, we only want to call functions of the class. Is there a way for us to return each variable as we did for the color attribute without directly accessing it? What code would we need to add?

Now in R we can create a similar class using this syntax (based off of these examples),

```
carMaker <- function(color, make, model, gas){
    exampleCar <- list(Color = color, Make = make, Model = model, Gas = gas)
    class(exampleCar) <- append(class(exampleCar), "car")



    return(exampleCar)
}
```

Here we’ve actually created a function that takes in arguments for our color, make, etc and gives us back a car class. We can create one of these cars and access its variables as follows,

```
# Create a new car
ourCar <- carMaker("Red", "Lamborghini", "Gallardo", 5)

# See the color of our car
ourCar$Color
```

```
## [1] "Red"
```

```
# Add gas
ourCar$Gas = ourCar$Gas + 4
ourCar$Gas
```

```
## [1] 9
```

So we have now modularized our code. We have objects that hold attributes (they can also hold functions). Ideally we would like to write code in which objects communicate with each other and the details of each are abstracted away. This means that if I have an object called attendant that fills the gas in the car; that object doesn't need to know how that gas is filled as long as the desired result is achieved. That means maybe the attendant wants to add 4 gallons of gas, and maybe the function multiplies this amount by 50 then divides by 50 then adds the gas. Although this logic is unnecessary it still gives the desired result of adding 4 gallons of gas. This is the heart of abstraction and we'll call these separations in code, abstraction barriers. This example was easily added to our Java code as a function that is internal to the class. With the R class we explicitly changed the variable gas itself. This breaks the abstraction barrier. We can create methods that accept a car and then extract these variables but there is still a separation between the encapsulated class and the external function.

Now the question, which language is better for OOP? While everything in R is technically an object and nearly everything in Java is an object, Java seems to take the cake here. Java allows for an easy way to create complex classes and to keep them encapsulated. Without a way to explicitly code a reusable class, R has created a way not only for abstraction barriers to be broken, but also for it to be easy to break them as your code evolves.

OOP is an extremely important programming concept. To learn more about it and the history behind this programming movement check out the wikipedia page.

Vectors and Arrays

Note: R also has what are called Arrays but they are not analogous to the data structures below.

Java has many more complex data structures at it's disposal but in todays world R is often run on distributed systems for large data processing which means if you have enough money you technically don't need to have the most efficient data structures. Since this is beyond the scope of this post we'll just focus on the most basic data structure, arrays/vectors. We're interested in how they work, how fast are they for a single machine, and how easy are they to interface with as a programmer.

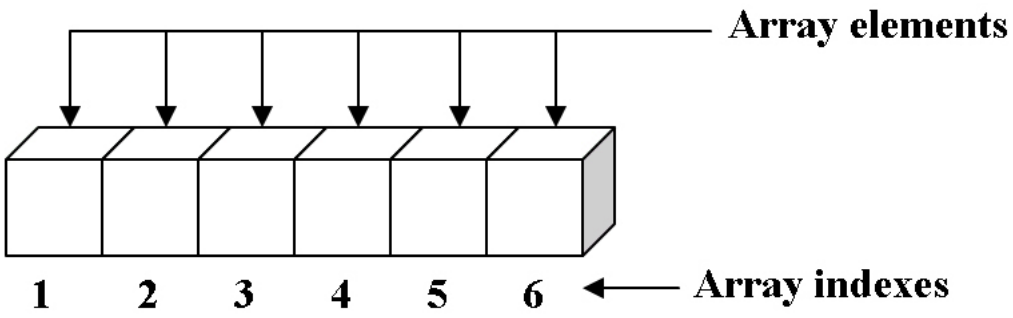
In R the lines become a bit blurred on what a vector can do as things get more complicated (some more complex data structures in R are actually vectors) but here we'll focus only on those of the type,

```
ourVector <- c(1, 2, 3, 4 ,5 ,6)
```

In Java this is analogous to,

```
int[] ourArray = new int[]{1, 2, 3, 4, 5, 6}
```

To understand what's happening we can look below,



One-dimensional array with six elements

Each of the entries in our array of vector can be thought of as being stored in a structure like this. The indexes below allow us to grab the entry at that index. In both languages, indexing is done with `ourArray[x]` where `x` is our index. The difference here is that unlike the image above, Java indexes starting at 0. The majority of programming languages actually start at 0 which can seem non intuitive for new users. The reasons however become clearer as one learns about computer architecture. The gist is that every one of those cells has its own address in your computers memory and each cell is size `n` where `n` in the size of in this case an integer (32 bits). This means that each cells memory address is evenly spaced, `n` apart. The computer always just holds the address of the first element and then to get another elements address in the array it can just multiply the index `x` by `n` and add it to the initial first address. If this doesn't make sense don't worry, it's inconsequential since starting with 1 instead of 0 only means an extra subtraction of 1 to get our element. The reasons to index at 1 are apparent to a human yet many programmers have other reasons to index at 0. Overall choosing what value to start indexing at doesn't matter as long you stay consistent.

If you're interested in pointer arithmetic and how computers work under the hood, check out this link that the ideas above are based on.

The biggest difference here, and one where R really shines is vector operations. Vector operations in R allow us to do one operation that is then done on all of our vector at once.

Lets look at how we'd subtract 1 from our Java array above,

```
for (int i = 0; i < ourArray.length; i++) {
    ourArray[i] = ourArray[i] - 1;
}
```

Here we are looping the same operation starting with an integer i at 0; using it to index and do our operation; then incrementing it; and starting again until we've gone through the whole array.

In R we can do this as,

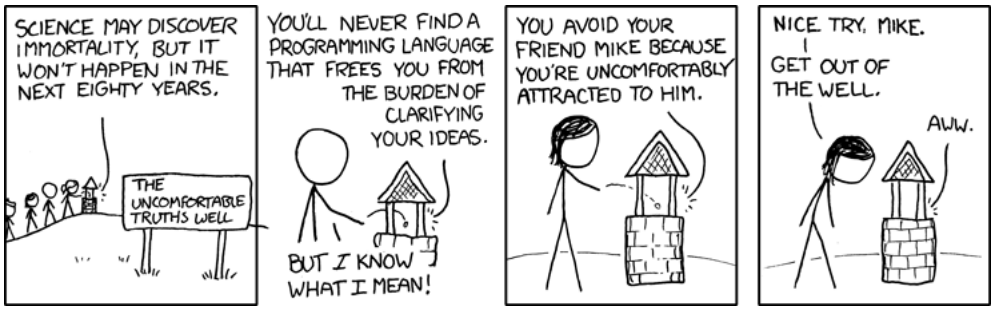
```
ourVector <- ourVector - 1
print(ourVector)
```

```
## [1] 0 1 2 3 4 5
```

This operation will remove one from every element in the vector without needing the loop (R does have loops if you are so inclined). A big thing here is that even though we are just writing it once, the computer still needs to go to each of those cells and remove 1 from them. Underneath this could be implemented in R as just a loop as in the Java example or it could take advantage in cases like this of the fact that we can remove 1 from the cells in any order and it won't affect each other, this means we can parallelize these operations and get massive performance boosts.

As a fan of indexing at 0 myself only due to my background I thought I would've given Java the win in this standoff but when it comes down to it the indexing pales in comparison to the performance gains made by abstracting vector operations away. This round goes to R.

Syntax, Ease of Use, and Elegance



This section is mostly a personal preference thing but I'll try to find some objective meaning. The first thing that comes to mind when thinking of these topics in regards to a programming language is readability. How easy is it for us to decipher whats going on? On one hand R doesn't use semicolons to delimit new lines, but on the other hand R has `<-` and `=` for assignment. On one hand Java has static types so every variable is labelled but on the other hand, function signatures can become long and convoluted. I think both languages have their pros and cons. Certain things like brackets and colons that may mess up new programmers can aid more experienced ones while the opposite is true in many other cases. I think both languages are equal in readability. With that said they both have very different ways of writing things so choose the language that works better with how your mind works.

Given the proper training, both languages can be taught easily and they both have a large amount of documentation online. Java has more documentation and far more content across the web but the language itself also has more features to warrant more documentation.

And finally when we think of elegance, whose to say? Vector operations in R are about as elegant as programming gets but the encapsulation and modularization of Java allows for orchestras of programs. Elegance is less about the language and more about the programmer. Your code will match your abilities and over time you will learn to write elegant code, no matter the language. As a programmer stackoverflow will become your main resource so to start you on your journey here is a link to a long discussion about how to write good code.

So who wins?

In the end it seems that no conclusion can fully be drawn for whether R or Java is better overall. Really there was no competition. The better language is the language that fits the current problem best. It's just a tool.

Both languages exist because people need them for different problems. You wouldn't write the backend for a web server in R and you wouldn't do big data analysis with Java. Part of becoming a competent programmer is knowing when to use what technology. If you learn your fundamentals when it comes to programming you'll be able to pickup whatever language you want.

So, what have we learned? We're comparing apples to oranges and we should just go for the one we crave. If you're a new programmer and you're now unsure of which tool to learn programming in, here's a tip, use Python □□.

References used throughout post:

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

https://www.tutorialspoint.com/r/r_data_types.htm

https://www.quora.com/What-is-OOP-with-examples

http://www.cyclismo.org/tutorial/R/s3Classes.html

http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BitOp/pointer.html

https://softwareengineering.stackexchange.com/questions/61655/how-do-you-know-youre-writing-good-code

http://www.steves-internet-guide.com/binary-numbers-explained/

https://en.wikipedia.org/wiki/Object-oriented_programming

Citations:

https://brilliant.org/wiki/arrays-adt/ for array image

https://xkcd.com/