Linguistic Parasitism

António Menezes Leitão

March, 2021

1 Introduction

Nowadays, programmers expect programming languages to provide a vast set of pre-defined libraries. However, it is important to note that these libraries require either huge finantial investments (e.g., the Java case) or hordes of programmers willing to develop those libraries for free (e.g., the Python case). This makes it very difficult for new programming languages to establish themselves in the software development arena.

When there is no money available nor the vast number of altruistic programmers needed to create the libraries, the next best option for a new programming language is to become a parasite, i.e., to take advantage of the libraries available in other programming languages. These languages will then become hosts for the parasite.

Unfortunately, this is not a trivial endeavor, as the parasite language and the host language might be very different, with different programming paradigms, different data structures, different function or method call capabilities, etc. As a result, in order to facilitate the use of the libraries available in the host language it is necessary to develop mechanisms that integrate well with the paradigms of the parasite language.

As an example, when the Scala language was proposed, it could directly use all the libraries available for the Java language. A similar approach was used for the Clojure language. In both cases, the parasitic strategy allowed the language to be immediately useful, giving it time to gain critical mass, to the point where language-specific libraries were proposed to replace the parasitic use of Java libraries.

As another example, when the Julia programming language was developed, the language designers considered, from the beginning, that it would be necessary to support the use of libraries written in the languages C and Fortran. These were important because, in the beginning, the language was mostly focused in scientific computation. As the range of application increased, it became obvious that there were numerous other useful libraries for Julia not written in C or Fortran. Instead, they existed in the Java ecosystem and, thus, it became important for Julia to parasitize Java as well. Unfortunately, Julia and Java are very different languages:

- Java is an object-oriented language, while Julia is mostly functional.
- Java is statically typed, while Julia is dynamically typed.
- Java uses single inheritance of classes and multiple subtyping through classes and interfaces, while Julia uses single subtyping of abstract types and composition of concrete types.
- Java uses method calls with dynamic dispatch on the receiver but static dispatch on the arguments, while Julia uses generic function calls with symmetric multiple dispatch on all arguments.
- Java's methods belong to classes, while Julia's methods belong to generic functions.
- Java indexes start with zero, while Julia prefers to start with one, although it supports other indexing schemes.

The list of differences is, obviously, much longer but these are enough to illustrate the main challenge of the parasitic approach: take advantage of the host without forcing the parasite to lose its identity.

As an example of the challenge, consider the Julia package **JavaCall** (`https://juliainterop.github.io/JavaCall.jl/`) that supports the use of Java libraries in Julia. Here is an interactive example of its capabilities, where we use Java to compute $\sin(\pi/2)$:

```
julia> using JavaCall

julia> JavaCall.init(["-Xmx128M"])

julia> jlM = @jimport java.lang.Math
JavaObject{:java.lang.Math} (constructor with 2 methods))

julia> jcall(jlM, "sin", jdouble, (jdouble,), pi/2)
1.0
```

Note, in the previous REPL example, that JavaCall uses the primitive operation `jcall` to make method calls. Unfortunately, this operation requires the specification of the method signature, i.e., name, return type, and parameter types. This makes it difficult to make even simple calls and, clearly, it does not integrate cleanly with Julia.

Fortunately, it is not difficult to make the Java library easier to use. We just need to define a function, which we will call `jsin`:

```
julia> jsin(x) = jcall(jlM, "sin", jdouble, (jdouble,), x)

julia> jsin(pi/2)
1.0
```

Another option is to preserve the Java syntax, e.g., by using a named tuple to store the corresponding functions:

```
julia> Math=(sin=x->jcall(jlM, "sin", jdouble, (jdouble,), x),
             cos=x->jcall(jlM, "cos", jdouble, (jdouble,), x))
(sin = var"#9#11"(), cos = var"#10#12"())

julia> Math.sin(pi/2)
1.0

julia> Math.cos(pi)
-1.0
```

It is obviously impractical to manually define functions for each and every Java method, but nothing prevents us from doing that automatically, through reflection and metaprogramming.

The previous examples illustrated the use of static methods but the idea also applies to non-static methods. The next example illustrates the use of Java's `LocalDate`:

```
julia> jtLD = @jimport java.time.LocalDate
JavaObject{Symbol("java.time.LocalDate")}

julia> local_date_now() =
         jcall(jtLD, "now", jtLD, ())
local_date_now (generic function with 1 method)

julia> local_date_now()
JavaObject{Symbol("java.time.LocalDate")}(
  JavaCall.JavaLocalRef(Ptr{Nothing} @0x00000000027ae060))
```

Note that the previous invocation returned a Julia value that operates as a *proxy* to the corresponding Java object. In order to provide a better presentation, we can specialize Julia's generic function `show` so that it calls Java's `toString` method.

```
julia> Base.show(io::IO, obj::JavaObject) =
         print(io, jcall(obj, "toString", JString, ()))

julia> local_date_now()
2021-04-27
```

Obviously, more Java methods can be defined as Julia functions:

```
julia> plus_days(jtld, days) =
         jcall(jtld, "plusDays", jtLD, (jlong,), days)
plus_days (generic function with 1 method)

julia> plus_days(local_date_now(), 4)
2021-05-01
```

As we mentioned before, it quickly gets annoying to have to manually define all of those functions so that is one area where a lot of improvement can be done.

Although the previous examples presented a functional approach to Java method calls, it is possible to present a more traditional object-oriented notation. Here is one possibility:

```
julia> struct JavaValue
         ref::JavaObject
         methods::Dict
       end

julia> Base.show(io::IO, jv::JavaValue) =
         show(io, getfield(jv, :ref))

julia> Base.getproperty(jv::JavaValue, sym::Symbol) =
         getfield(jv, :methods)[sym](getfield(jv, :ref))

julia> jtLDMethods = Dict(
         :plusDays => (jtld) ->
           (days) ->
             JavaValue(jcall(jtld, "plusDays", jtLD, (jlong,), days),
                       jtLDMethods),
         :plusMonths => (jtld) ->
           (months) ->
             JavaValue(jcall(jtld, "plusMonths", jtLD, (jlong,), months),
                       jtLDMethods),
         :plusWeeks  => (jtld) ->
           (weeks)  ->
             JavaValue(jcall(jtld, "plusWeeks", jtLD, (jlong,), weeks),
                       jtLDMethods),
         :plusYears  => (jtld) ->
           (years)  ->
             JavaValue(jcall(jtld, "plusYears", jtLD, (jlong,), years),
                       jtLDMethods))
Dict{Symbol, Function} with 4 entries:
  :plusYears  => #19
  :plusDays   => #13
  :plusMonths => #15
  :plusWeeks  => #17

julia> now = JavaValue(local_date_now(), jtLDMethods)
2021-04-28

julia> now.plusDays(4)
2021-05-02

julia> now.plusWeeks(2).plusDays(4)
2021-05-16
```

Again, most of what was done manually could have been done automatically. Obviously, what we interactively ilustrated using the Julia REPL is not production-quality code and it needs to be considerably improved.

2 Goals

The main goal of this project is to make the Julia programming language capable of using Java libraries in the nicest way you can imagine. That is all!

2.1 Extensions

You can extend your project to further increase your final grade. Note that this increase will not exceed **two** points.

Examples of interesting extensions include:

- Integrating the Java exception mechanisms into Julia.
- Being able to define, in Julia, subclasses of Java classes.
- Allowing callbacks from Java to Julia, e.g., using Julia's anonymous functions.
- Solving Java's constructor madness.

Be careful when implementing extensions, so that the extra functionality does not compromise the functionality asked in the previous sections.

3 Code

Your implementation must work in Julia 1.6 and Java 8 but you can also use more recent versions.

The written code should have the best possible style, should allow easy reading, and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided into functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

4 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 10-minute slot, should be centered on the architectural decisions taken, and may include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers. You should be prepared to answer questions regarding both the presentation and the actual solution. If necessary, you will be asked to discuss your work in full detail in a separate session.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- The source code, within subdirectory `/src/`

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `presentation.pdf`.

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the healthy exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 23:00 of **June**, **4**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **June**, **4**.

The presentations will be done during classes after the deadline. Only one element of the group will present the work. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may ask questions to any member of the group before, during, and after the presentation.