Why do languages come with modules, packages, or namespaces?

1 The Pragmatics Question

The models of the first six chapters roughly correspond to the programming languages people invented and used from the 1950s through the 1970s. Back then, the creation of software was mostly the task of a lonely programmer or small teams. In-person meetings established conventions and protocols that overcame weaknesses in programming-language support of separate development of components.

Imagine the following scenario in such languages:

- All programmers need libraries beyond the basic ones that a sparse language provides. They may want basic data structure libraries or basic mathematical algorithms. Eventually someone creates such libraries, say in the form of classes. The language implementation is then modified so that it *prepends* these libraries to the programmer-developed code as part of the loading process.

- A manager designs a rather large software system and recognizes that this is way too large for the current team. So the team is charged with identifying the interfaces of components, say collections of yet-to-be-implemented classes, and to divide the work among the members of the team and other teams. For a time, all these people work separately, creating classes to meet the interface specifications and auxiliary classes to factor these implementations into reasonable chunks. The manager's idea is to *append* all these separately developed classes to set up and run one complete program.

If all this is done naively, some such program is sooner or later going to suffer from name clashes.

A team using a plain class-based language could work around this shortcoming with the use of naming protocols. Each member would have to systematically use this protocol—manually. Perhaps the team could also agree to use an IDE tool to make sure the code adheres to this protocol. However, any work situation that involves some kind of manual checking—even with an IDE tool—is highly error-prone. Just imagine a team that produces the first version of a successful system whose members quickly quit one after another. New members are brought on. Everyone needs to be trained in the use of the protocol and the IDE tools. Sooner or later, someone is going to violate the protocol and cause havoc.

In our context, support for software development in a team means a mechanism for dealing with separately developed classes. That is, we imagine that a language design that would enable a team to divide up a system design into separately implemented pieces. Each of these pieces may contain a class or even several, and each of these depends only on a few, explicitly declared such pieces. As a result, these pieces can be re-used in many systems; they become library components.

1.1 Design Choices

Programming language creators recognized the problem as early as 1980. Since then they have created a fair number of linguistic mechanisms to address the naming problem:

- *Modula*, an aptly named language, introduced the idea of a module into the design space around 1980.

- The family of Lisp languages started supporting namespaces around the same time. From there the idea spread to other languages, including Java.

  The names of classes, variables, and fields exist in separate namespaces, and the Java implementation resolve the same name in these distinct spaces. Consider figure 41, which presents two classes: Main and c. While the latter also contains a field named c and uses c as a parameter name in the constructor, the former creates an instance of c, stores it in the variable named c, and then access the field of this instance via c..

  Exercise 42. Compile and run the Java program in figure 41. It is going to print 66, which confirms that the Java implementation is capable of managing the various roles of the name c.

- Class-based languages added packages and inner-classes to address the issue. Both enable software developers to manage names, usually via the use of so-called fully qualified names.

  A package in Java is a container for related classes. For example, java.util.Date is a package that offers a generic data representation for dates and pieces of functionality for manipulating it. And, java.sql.Date is a package for representing and manipulating SQL dates, that is, dates as used in the database language.

  By now, though, even Java also offers modules. Its designers have understood that classes, packages, and modules play distinct roles in different working situations.

```
class Main {
  public static void main(String argv[]) {
    c c = new c(66);
    System.out.println(c.c);
    return ;
  }
}
```

```
class c {
  int c = 2;
  c(int c) {
    this.c = c;
  }
}
```

Figure 41: Java: namespaces and fully qualified package names

This chapter presents a simple model of module-based extension of *Class*. Its purpose is to bring across how modules work, how they differ from classes, and how they assist software developers with their work in teams. For now, imagine the following general arrangement:

- We use the word *system* to refer to a collection of modules.
- A module contains and exports class definitions.
- A module may import all classes from already existing modules.
- Since it is about name management, a *Module* programmer should recognize the underlying class-based program without any problem.

Like all models, this one greatly simplifies actual module mechanisms. Most languages offer all kinds of conveniences on top of these basic aspects. But, once you understand the model presented here, you can work out the additional costs and benefits of them on your own.

1.2 Costs and Benefits

A system built from modules provides two major advantages for some immediate but small notational overhead. The key is that each module isolates a developer's work from the work of others while simultaneously making its dependencies on other modules explicit. In turn, the language implementation confirms the correctness of the dependency specifications, which enables it to compile modules independently. To receive these benefits, a developer must add the lines of code that state the module-level dependencies; fortunately, modern IDEs can assist developers with this task.

The picture differs for the language creator. It starts with the idea that modules are just textual containers that isolate one bunch of classes from others. As such, they are syntax without any meaning—or as implied so far, a system of modules imposes organization on an executable class-based program. To ensure that this program is well-formed and valid, the language creators have to work out appropriate validity constraints on the system as a whole and individual modules.

While these validity constraints aren't complex, their existence has implications for the developer as well as the language implementer. For the former, it steepens the learning curve. Concretely the developer must continue to respect all the existing validity constraints for the underlying language of classes and must learn what it means to formulate modules properly. The language implementer has to add code to the existing validity checker so that these module-level validity constraints are enforced. When the checker discovers violations of the constraints, it must synthesize a reasonable amount of information so that the IDE can inform the developer.

Finally, the language implementer must supply a linker. Roughly speaking this *linker* is the tool that unwraps the modules and creates a plain class-based program. Unwrapping means class definitions are directly exposed to each other. Since the introduction of modules is partly about freeing team members from having to adhere to naming protocols for classes, a naive unwrapping might cause undesirable name clashes. One way to eliminate this problem is to shape the module level so that such clashes cannot occur. If this is possible, the linker always succeeds in creating a well-formed and valid class-based program—meaning the tool itself does not impose any intellectual cost on the working software developer, at the cost of having to understand the rules of modules.

Similarly, this arrangement also has a benefit for the language implementer. The linker is going to create a class-based program that can run on the existing CESK machine. In other words, the module system does not

require any changes to the rather complicated and subtle machinery that assigns meaning to programs and thus systems.

2 Project Language: *Module*

Like all of our models, *Module* presents the bare minimum of a language with modules. Here are the model's essential characteristics:

- A *Module system* consists of a sequence of module definitions, followed by a *Core* program with references to some of the modules.
- A module definition introduces a name and contains *a single class* definition, which is also visible to the outside—i.e. is *exported*.
- Each module may refer to classes from modules that precede it in the sequence of modules, that is, *import* existing modules.

As usual, the model is small and omits many features of real-world languages that support modules. The goal is to understand the basics of name management and how it merely imposes structure on a *Class* program, not adding any semantics. If you'd like to explore additional features, feel free to add and study them on your own.

2.1 *Module*: the Grammar

Figure 42 presents the BNF grammar for the syntax of *Module*. The first production shows how *Module* systems are created. They consist of a potentially empty sequence of modules, a potentially empty sequence of import specifications, followed by a *Core* program—referred to as the *system's body* in this chapter.

The second production introduces the rather simple module concept. A module has a name and contains (1) a sequence of import specifications and (2) a single class definition. The third and last production introduces import specifications. Each import names a single module; its purpose is to make the respective class definition available in the scope of the module.

Intuitively a module is a container that protects the class definition from name interference. Only imported ClassNames are visible, and exactly one ClassName is exported. Otherwise the code inside of a module must obey the grammatical rules of *Class*.

```
System      ::= (Module*
                 Import*
                 Declaration*
                 Statement*
                 Expression)

Module      ::= (module ModuleName Import* Class)

Import      ::= (import ModuleName)


The set of ModuleNames is the same as the set of Variables.
```

Figure 42: The grammar of the *Module* language

A system consisting of modules also directly corresponds to a *Class* program if we think of module names as prefixes for class names. Consider the sketch of a *Module* system on the left:

Java uses `$` for similar purposes. Many other languages come with similar conventions.

a *Module* system	the corresponding *Class* program
`(module M (class C ... ))`	`(class M.C ... )`
`...`	`...`
`(module K (class C ... ))`	`(class K.C ... )`
`...`	`...`
`(module L`	
`  (import K)`	
`  (class D ... C ...))`	`(class D ... K.C ...))`
`...`	`...`
`(import M)`	
`... C ... C ...`	`... M.C ... M.C ...`

Module M as well as K contain a class definition named C. But, as the *Class* program on the right shows, if we loosen the rules for formulating class names—by allowing the dot between letters—the translation is straightforward and resolves any doubts about which C is meant in which context. In short, our model of a module language satisfies all four desiderata spelled out in the introduction of this chapter, though as usual, in a highly simplified manner.

Exercise 43. Design an AST data representation for *Module*. Implement a parser for *Module* that maps an S-expression to an instance of AST. Re-use your solution from exercises 10 and 17 as much as possible.

2.2 *Module*: Scope

Once a language designer has finished a (first draft of a) grammar, the next step is to specify the scope of names. That is, for each kind of name, it is necessary to spell out the lexical regions where its visible. Based on a full understanding of scope, we can state and impose the validity constraints. In the case of *Module*, it also enables us to design the linker, which turns a system into a *Class* program.

The entire system consists of two lexical regions: the sequence of modules and the system's body. In the former, the nth module binds its name in the remainder of the sequence of modules, starting at module n+1, and the system's body. The names of all modules are available, though not necessarily mentioned, in the import specifications of the system's body.

Next, any sequence of import specifications sets up a lexical scope. An individual import binds the ClassName of its module in the the remainder of the sequence plus the module's class definition or the system's body, respectively. If an import binds a ClassName that already belongs to the set of bound names, it shadows (in the sense of Chapter IV) the existing one.

In this spirit, each individual module is a lexical region, too. In this region, a ClassName must refer to the internally defined class or to the names of imported classes. Otherwise, name references are resolved in this region just like in an ordinary class definition à la *Class*.

Finally, the system's body forms a scope. In this region, a ClassName must refer to one of the names of imported classes. And again, otherwise the ordinary scoping rules of *Core* and *Class* apply.

Exercise 44. Draw the lexical region for each binding occurrence of a name in the following program:

```
(module M (class C ... ))
...
(module K
  (import M)
  (class C ... C ... ))
...

(import K)
(import M)

... C ... C ...
```

The last point to clarify is how ModuleNames, ClassNames, and variables co-exist in *Module*. In *Class*, a ClassNames is already somewhat like a variable and somewhat special. Consider this simple *Class* program:

```
((class C ()
   (method m(C)
     (new C (C))))
 (def c (new C()))
 (c --> m(c)))
```

Here C is the name of the only class, and it is the name of method m's only parameter. The scoping rules of *Class* demand only that C is a defined class in an expression such as (new C ...), which it is. Similarly, the scoping rules of *Class* require that a name in a constructor expression such as (new ... (C)) refer to a defined variable, which in this example it does. If so, the CESK machine for *Class* can evaluate such an expression without any problems.

What this admittedly odd example shows is that the *Class* model implicitly introduced Java-style namespaces: one for ClassNames and one for ordinary variables. Indeed, arguably the names of fields and methods form a third namespace, though in terms of scope this space is irrelevant.

Our *Module* model adds yet another namespace, namely, the names of modules. Fortunately, all these spaces are easy to keep separate because there are only a few places where most names may appear:

- ModuleNames set up a region when they are used in module definitions, and they show up only in import specifications are references to (existing) modules.

- ClassNames are used to name a class, and references to class names show up in just two expressions: new and isa.

- All other names, with the exception of FieldNames and MethodNames, which aren't scoped, denote ordinary program variables.

The existence of separate spaces of names is a convenience, not a necessity. Our model would work equally well if all names existed in just one space.

Exercise 45. Draw the lexical region for each binding occurrence

```
((module A
  (class A (A)
    (method A (A)
      A)))

 (import A)
 1.0 )
```

2.3 *Module*: Validity

A validity checker enforces scoping rules. That is, it is a translation of the rules into an algorithm that works for every well-formed module system. For the models of the preceding chapters, this translation is a straightforward process, because the scoping rules spell out explicit constraints. In the case of *Module*, we will need to make some implicit assumptions explicit.

For example, the preceding section implies that two sequences of name-binding linguistic constructs call for two distinct treatments. On one hand, the sequence of modules sets up nested scopes so you might think that it should be treated like a sequence of variable declarations. Concretely, if two modules have the same name, the second one in the sequence shadows the first one. But, consider this sketch of a module sequence and its corresponding program:

A *Module* system fragment	... and its "unwrapping"
`(module M (class C ... ))`	`(class M.C ... ))`
`...`	`...`
`(module M (class C ... ))`	`(class M.C ... ))`
`...`	`...`

A straightforward unwrapping of this system yields an invalid *Class* program. The unwrapping, and thus the semantics, of module systems would have to become a sophisticated process. We therefore rule out that the system may contain two modules with the same name.

Note The simplicity of the model makes the constraint look silly. It seems like we lifted the unique-name constraint from the class level to the module level. Keep in mind, though, that a real module, or say package, would contain many classes. Hence, the constraint is truly a help to teams who split up their work via interfaces, each of which may demand a realization with many classes.

On the other hand, a sequence of imports, which also forms a sequence of nested scopes, can easily accommodate the injection of two classes with identical names. The nesting tells us that the second one shadows the first. In short, importing two classes with the same name might call for a warning but it is consistent with the informal ideas behind our model.

Exercise 46. One aspect of managing names in *Module* is to allow developers to use the *same* name for different classes in distinct modules. Unlike in *Module*, though, existing programming languages allow programmers to import and *use* a class into a module even if it has the same name as one of

the internally defined classes. They support this name resolution via a notation that distinguishes the imported class from the internal one. Design such a notation for *Module*.

Putting these two thoughts together suggests that the *validity checker* for *Module* has to enforce the following two constraints:

- No two modules in a system may have the same ModuleName.
- Every import specification must refer to a ModuleName that is defined in the preceding sequence of modules. This includes the import specifications in the body of the system.

In addition, the specification of scoping rules impose two more constraints:

- Each module must be closed.
- The system's body must be closed.

Recall that "closed" means all variable occurrences point back to a specific declaration or a method's parameter. Furthermore, all ClassNames refer to the module-defined class or an imported class.

Exercise 47. Design and implement a validity checker that enforces the validity rules for *Module*. The checker consumes error-free ASTs from exercise 43; if it finds errors it annotates the AST appropriately.

3 Linking Modules into Programs

The language-related software tools of the preceding chapters fall into two classes: those related to syntax and others related to semantics. A parser or a validity checker belong to the first kind. A load function or a runMachine function belong to the second kind.

By contrast, a linker is neither a syntax-related tool nor a semantics-related one. It exists to related ASTs for one language, *Module*, to ASTs of a second one, *Class*. The point of such a translation is to explain the meaning of the first kind of AST via the second kind. From the perspective of a language creator or a model maintainer, the linker avoids the need to revise the intricate workings of the CESK machine for *Class*.

The functional purpose of a *linker* is to translate a well-formed and valid *Module* system into (the AST of) a well-formed and valid *Class* program. As indicated already, this translation is accomplished via a two-step transformation:

- First, the linker replaces *all* occurrences of ClassName with their fully-qualified names:
  - If class C is defined inside of module M, then its fully-qualified name is M.C.
    As mentioned in the preceding section, this name that is illegal in the surface syntax, but since the linker operates on ASTs, we can accept this loosening of constraints on acceptable names.
  - The replacement of the defining ClassName is easy.
  - The replacement of any other occurrence must account for its binding.
- Second, the linker strips the module and import forms from the AST.

The resulting *linked* program is well-formed and valid because each module contains a single class, meaning any two classes have distinct names and all other names are left alone.

a *Module* system	the corresponding *Class* program
`(module M (class C ... ))`	`(class M.C ... )`
`...`	`...`

`(module K`
`  (import M)`
`  (class C ... C ... ))`
	`(class K.C ... K.C ...)`

`...`	`...`

`(module L`
`  (import M)`
`  (class D ... C ... D))`
	`(class L.D ... M.C ... L.D))`

`(import K)`
`(import M)`
`... C ... C ...`
	`... M.C ... M.C ...`

Figure 43: Linking a *Module* system into a *Class* program

Figure 43 displays a system fragment and its corresponding program. It illustrates a couple of key properties of the linking process. First, every definitional name of a class is prefixed in the process. Second, if a module such as K imports a class that has the same name as the internally defined one, all occurrences of this ClassName point back to the local one in the resulting program. In contrast, if the imported class has a different name,

as is the case for module L, the imported class is visible via its fully qualified name. Finally, the system's body resolves class names according to the scoping rules concerning sequences of import specifications.

Exercise 48. Design and implement a linker for *Module*. The linker should map an AST of a *Module* system to a *Class* program. Use your solutions from exercises 43 and 47 and the project from Chapter VI.

4 Pragmatics: Computability vs Expressive Power

In the 1930s, Church and Turing settled on the hypothesis that all algorithmic languages can compute the exact same set of functions on the natural numbers, namely, the set of partial recursive functions. People such as Post, Smullyan, and some others tried to develop alternative computing systems, but all of them turned out to be equivalent to Church's lambda calculus and Turing's machines. Equivalent means that every algorithm in a computing system can be translated into an expression in an alternative one such that both provably compute the same function. From this perspective, all programming languages are the same.

Programmers know better, however. Instinctively they argue that some languages are "better" at some task than others. The simplest reason might be that a particular platform (hardware, browser) does not support some language, so another one must be used to implement an algorithm.

This chapter offers a natural example to ask the question in a scientific manner, because the two models—*Module* and *Class*—differ in just one way: the first has modules, the second doesn't. Hence the question is whether one can express "more" computations than the other. If so, we have a first idea of how to judge languages systematically.

Stop! Write a simple *Class* program that *Module* cannot directly express.

You may wonder what "directly" means in this context. Given our naive understanding of *Module* systems as *Class* programs and given the linker, which translate the former into the latter, the interpretation of "directly" should be straightforward: wrap each class definition in a module and add import specifications as needed.

Stop! Now that "directly" is understood, try your hands at the above exercise again.

Here is one such program:

```
(class C () (method m (x) (x isa D)))
(class D () (method m (x) (x isa C)))
(def c (new C()))
```

```
(def d (new D()))
(c --> m(d))
```

Due to global scope of the two class names—C and D—the two classes refer to each other in a valid manner. However, if we were to directly embed these classes into modules, the result would be a well-formed but invalid system:

```
(module M (class C () (method m (x) (x isa D))))
(module K (import M) (class D () (method m (x) (x isa C))))
(import M)
(import K)
(def c (new C()))
(def d (new D()))
(c --> m(d))
```

Since modules may import just those modules that precede them, it is impossible to resolve the (boxed) name D in M. Generally speaking, setting up mutually recursive classes is impossible in the module-based model language.

All programming languages come with such weaknesses, and when a language exhibits a weakness, developers create workarounds. In this case, the workaround consists of the definition of a third module that comes with the desired functionality:

```
(module K (class C () (method m (x o) (o --> isaD(x)))))
(module L (class D () (method m (x o) (o --> isaC(x)))))
(module M
 (import K)
 (import L)
 (class CD ()
  (method isaC (x) (x isa C))
  (method isaD (x) (x isa D))))

(import K)
(import L)
(import M)

(def c (new C()))
(def d (new D()))
(def cd (new CD())) ;; <--- an instance of CD
(c --> m(d o))      ;; <--- m is modified to accept cd
```

Once the system is equipped with this synthetic module and class, an instance of the latter can be created and passed along to method calls on instances of the original classes, C and D.

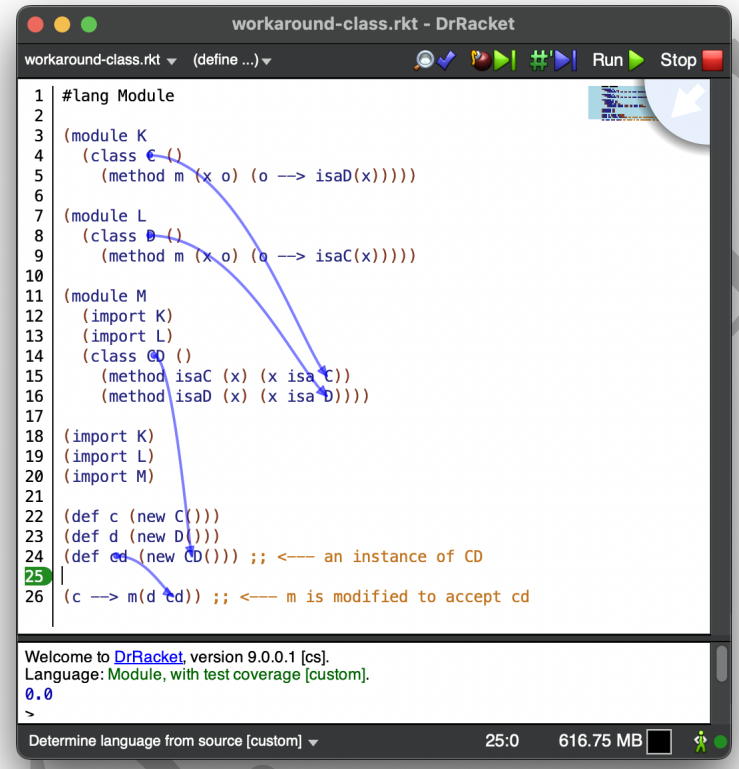

Figure 44: The indirections needed for the *Module* workaround

Taking a step back and a close look reveals the problem with workarounds. Our way of overcoming the weakness of *Module* is to impose a usage protocol on the entire system:

- Someone must create a synthetic module that can provides the needed functionality.
- The system's body must create an instance.
- Worst of all, the other team members must modify the method definitions and method calls so that, when needed, they accept and use instances of the synthetic module.

In short, the weakness of a programming language affects developers when they need a certain piece of functionality, and workarounds may impose a significant system-wide usage protocol with a non-trivial number of levels of indirections. See figure 44 for the latter.

The idea of comparing two languages that essentially differ in just one linguistic construct is due to Felleisen. It provides a well-founded argument for the rejection of the Church-Turing hypothesis as a relevant idea in the area of software development. For our purposes—understanding how programming languages (don't) support developers in certain work situations—it sets up a theoretical approach to evaluating the pragmatic information of a language construct. Sadly, it is a limited tool and further progress is needed to broaden this theoretical approach to pragmatics.

Note When Church and Turing claim that any two full-fledged languages are equivalent, they rely on translations that map code in one language to the other and vice versa. For their purposes, any translation from the *entire* space of functions between these languages is a good one. By contrast, Felleisen *restricts* the space of functions to those that require just local changes. This restriction corresponds to the idea that one team member cannot force all other team members to change their modules, classes, and methods just because some system-wide protocol would fix a programming-language weakness.