Modularization & ADTs

Readings: CP:AMA 19.1

Modularization

In previous courses we designed small programs with definitions in a single Racket (`.rkt`) file.

For larger programs, keeping all of the code in one file is unwieldy. Teamwork on a single file is awkward, and it is difficult to share or re-use code between programs.

A better strategy is to use ***modularization*** to divide programs into well defined **modules**.

The concept of modularization extends far beyond computer science. You can see examples of modularization in construction, automobiles, furniture, nature, etc.

A practical example of a good modular design is a “AA battery”.

We have already seen an elementary type of modularization in the form of *helper functions* that can “help” many other functions.

We will extend and formalize this notion of modularization.

When designing larger programs, we move from writing “helper functions” to writing “helper modules”.

A ***module*** *provides* a collection of functions† that share a common aspect or purpose.

† Modules can provide elements that are not functions (*e.g.*, data structures and variables) but their primary purpose is to provide functions.

For convenience in these notes, we describe modules as providing only functions.

Modules *vs.* files

In this course, and in much of the "real world", it is considered **good style** to store **each module in a separate file**.

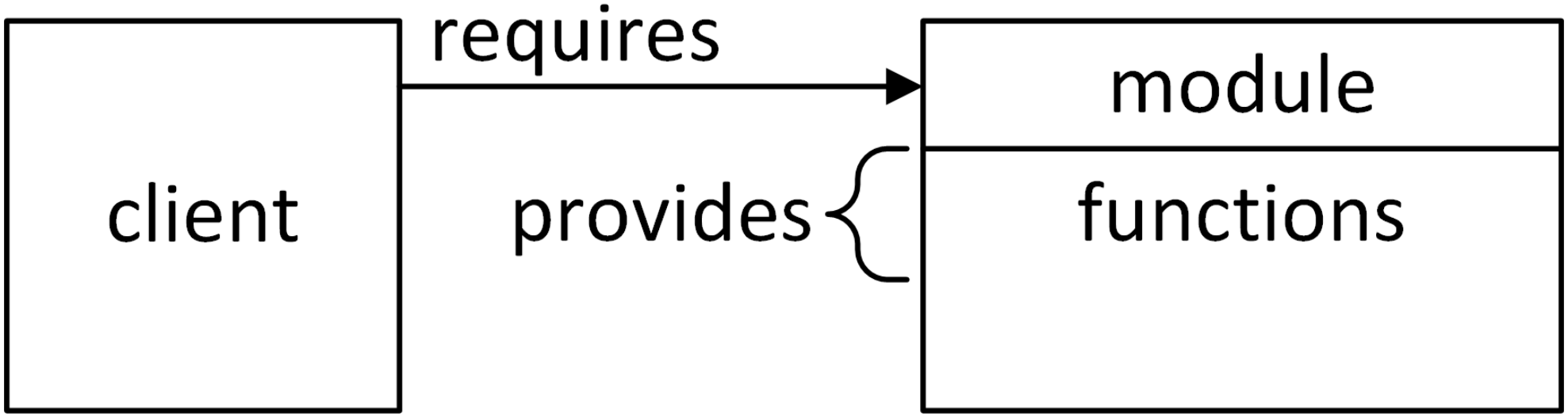
While the terms *file* and *module* are often used interchangeably, a file is only a module if it provides functions for use outside of the file.

Some computer languages enforce this relationship (one file per module), while in others it is only a popular *convention*.

There are advanced situations (beyond the scope of this course) where it may be more appropriate to store multiple modules in one file, or to split a module across multiple files.

Terminology

It is helpful to think of a "**client**" that **requires** the functions that a module **provides**.

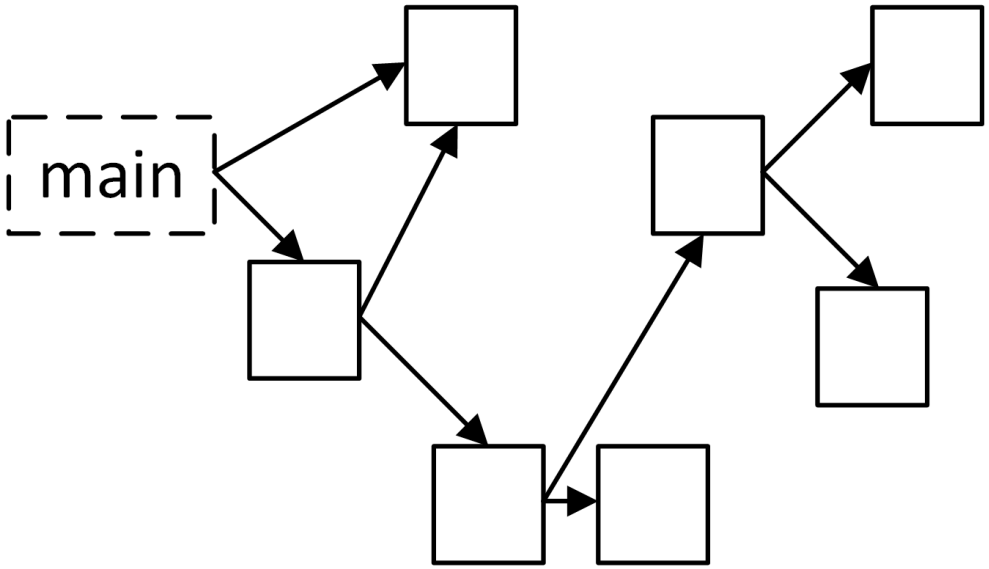


In practice, the client is a file that may be written by yourself, a co-worker or even a stranger.

Conceptually, it is helpful to imagine the client as a separate entity.

Large programs can be built from many modules.

A module can be a client itself and *require* functions from other modules.



The *module dependency graph* cannot have any cycles.

There must be a “root” (or ***main file***) that acts only as a client.

This is the program file that is “run”.

Motivation

There are three key advantages to modularization: re-usability, maintainability and abstraction.

Re-usability: A good module can be re-used by many clients. Once we have a "repository" of re-usable modules, we can construct large programs more easily.

Maintainability: It is much easier to test and debug a single module instead of a large program. If a bug is found, only the module that contains the bug needs to be fixed. We can even replace an entire module with a more efficient or more robust implementation.

Abstraction: To use a module, the client needs to understand **what** functionality it provides, but it does not need to understand **how** it is implemented. In other words, the client only needs an *"abstraction"* of how it works. This allows us to write large programs without having to understand how every piece works.

Modularization is also known in computer science as the *Separation of Concerns (SoC)*.

example: fun number module

Consider that some integers are more “fun” than others, and we want to create a **fun** module that `provide`s a `fun?` function.

```
;; fun.rkt

(provide fun?)  ; <---- this is new!

(define lofn '(-3 7 42 136 1337 4010 8675309))

;; (fun? n) determines if n is a fun integer
;; fun?: Int -> Bool
(define (fun? n)
  (not (false? (member n lofn))))
```

The `provide` special form above makes the `fun?` function available to clients.

The `require` special form allows clients to use the `fun?` function `provided` by the **fun** module.

```
;; client.rkt

(require "fun.rkt")  ; <---- this is new!

(fun? 4010)     ; => #t
(fun? 4011)     ; => #f
```

re-usability: multiple programs can use the fun module.

maintainability: When new numbers become fun (or become less fun), only the fun module needs to be changed.

abstraction: The client does not need to understand what makes an integer fun.

Modules in Racket

We have just seen the two Racket special forms that allow us to work with modules: `provide` and `require`.

See Appendix A.3 for more examples of working with `provide` and `require`.

Scope

Previously, we have seen two levels of identifier ***scope***:

- **local** identifiers are only visible inside of the `local` region (or function body) where it is defined
- **global** identifiers are defined at the *top level*, and are visible to all code following the definition

Because `local`s can be "nested" there can be multiple "levels" of local scope, but in this slide we are generalizing.

`provide` introduces a new level of scope.

Global (top-level) identifiers can have either **program *or* module** scope.

- **module** identifiers are only visible inside of the module (file) they are defined in
- **program** identifiers are visible outside of the module (file) they are defined in (*i.e.,* they are `provide`d)

We will continue to use *global scope* to refer to identifiers that have either program or module scope.

example: scope

```
;; mymodule.rkt

(provide pscope)

(define (pscope x)      ; program scope (provided)
  ... )

(define (mscope x)      ; module scope (not provided)
  (define lscope ...)   ; local scope (within a function)
  ...)
```

In addition, each parameter `x` has local scope.

Other courses may use different scope terminologies.

Module interface

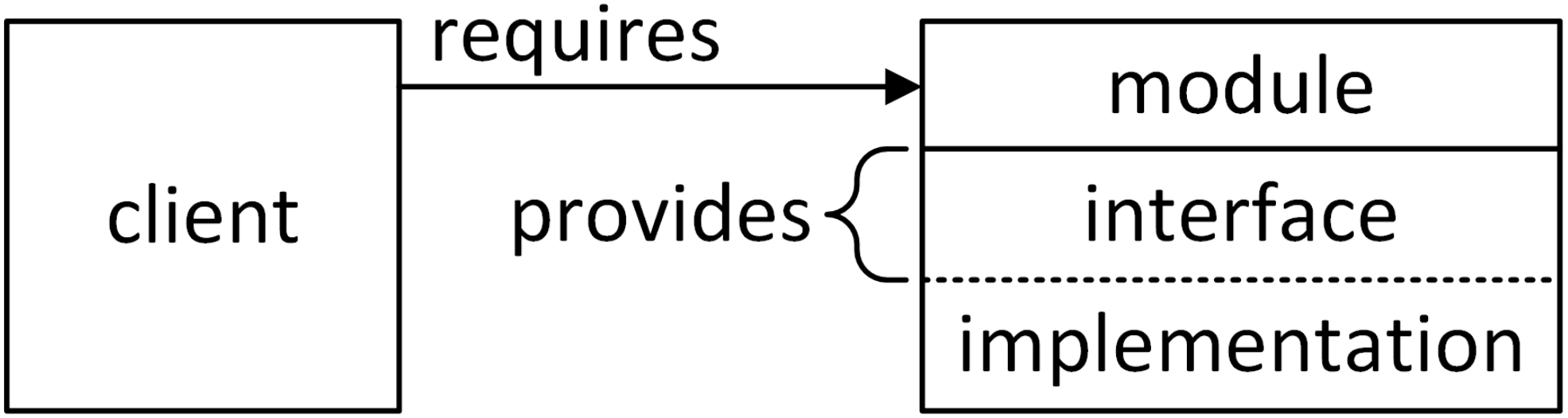
The module ***interface*** is the list of the functions that the module provides.

In practice, the interface also includes the documentation.

The interface is separate from the module ***implementation***, which is the code of the module (*i.e.,* function **definitions**).

The interface is everything that a client would need to use the module. The client does not need to see the implementation.

Terminology (revisited)



The **interface** is what is provided to the client.

The **implementation** is hidden from the client.

Interface documentation

Interface documentation includes:

- an **overall description** of the module,
- a **list of functions** it provides, and
- the **contract** and **purpose** for each provided function.

Ideally, the interface should also provide *examples* to illustrate how the module is used and how the interface functions interact.

Racket module interfaces

For Racket modules, the interface should appear at the top of the file above the implementation (function definitions).

In the implementation, it is not necessary to duplicate the interface documentation for public (program scope) functions that are `provide`d.

For private (module scope) functions, the proper documentation (contract and purpose) should accompany the function definition.

example: sum module

```
;; A module for summing numbers [description of module]

(provide sum-first sum-squares)  ;; [list of functions]

;; (sum-first n) sums the integers 1..n
;; sum-first: Int -> Int
;; requires: n >= 1

;; (sum-squares n) ...

;;;;;;;;;; IMPLEMENTATION ;;;;;;;;;;

;; see interface above  [no further info required]
(define (sum-first n) ...)

;; see interface above  [no further info required]
(define (sum-squares n) ...)

;; [purpose & contract for private helper function]
(define (private-helper p) ...)
```

Testing

For each module you design, it is good practice to create a **test client** that ensures the `provide`d functions are correct.

example: test-sum.rkt

```
;; this is a simple testing client for sum.rkt
(require "sum.rkt")

; Each of the following should produce #t
(equal? (sum-first 1) 1)
(equal? (sum-first 2) 3)
(equal? (sum-first 3) 6)
(equal? (sum-first 10) 55)
```

In Section 07 we discuss more advanced testing strategies.

There may be “white box” tests that cannot be tested by a client. These may include implementation-specific tests and tests for module-scope functions.

In these circumstances, you can `provide` a `test-module-name` predicate function that produces `#t` if the tests are successful.

Designing modules

The ability to break a big programming project into smaller modules, and to define the interfaces between modules, is an important skill that will be explored in later courses.

Unfortunately, due to the nature of the assignments in this course, there are very few opportunities for you to **design** any module interfaces.

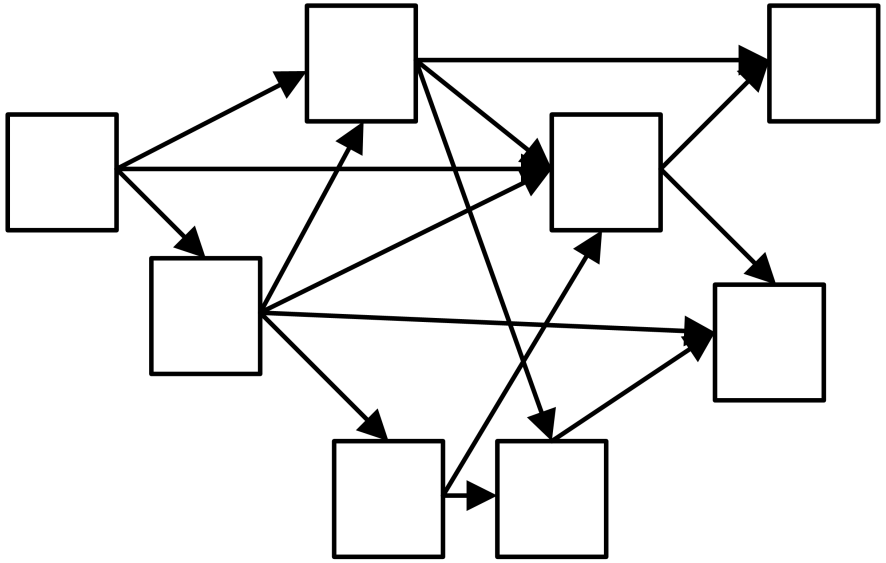
For now, we will have a brief discussion on what constitutes a **good** interface design.

Cohesion and coupling

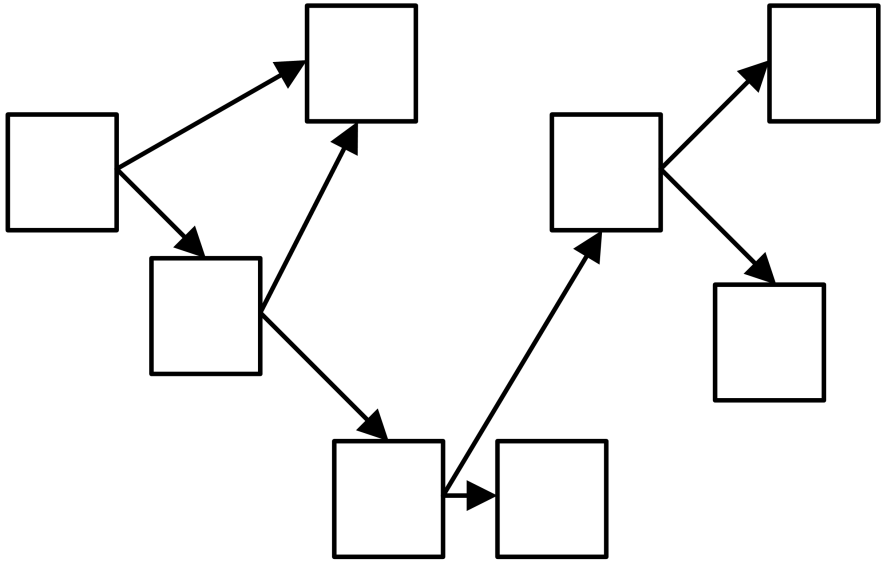
When designing module interfaces, we want to achieve ***high cohesion*** and ***low coupling***.

High cohesion means that all of the interface functions are related and working toward a "common goal". A module with many unrelated interface functions is poorly designed.

Low coupling means that there is little interaction *between* modules. It is impossible to completely eliminate module interaction, but it should be minimized.



High coupling



Low coupling

Interface *vs.* implementation

We emphasized the distinction between the module **interface** and the module **implementation**.

Another important aspect of interface design is ***information hiding***, where the interface is designed to hide any implementation details from the client.

In Racket, the module interface and implementation appear in the same file, so the distinction between interface and implementation may not seem important, but in C (and in many other languages) only the interface is provided to the client.

Information hiding

The two key advantages of information hiding are ***security*** and ***flexibility***.

Security is important because we may want to prevent the client from tampering with data used by the module. Even if the tampering is not malicious, we may want to ensure that the only way the client can interact with the module is through the interface. We may need to protect the client from themselves.

By hiding the implementation details from the client, we gain the **flexibility** to change the implementation in the future.

example: account module

We want to design a module that stores username and password data in an "account" and provides functions for retrieving the username and verifying a password.

```
;; account.rkt :: a module for managing an account

(provide create-account get-username correct-password?)

;; (create-account username password) creates an account
;;   with the given username and password
;; create-account: Str Str -> Account

;; (get-username acc) retrieves the username of acc
;; get-username: Account -> Str

;; (correct-password? acc word) determines if word
;;   is a valid password for acc
;; correct-password?: Account Str -> Bool
```

Consider the following naïve implementation that simply stores the data in a list with two elements.

```
(define (create-account username password)
  (list username password))

(define (get-username acc)
  (first acc))

(define (correct-password? acc word)
  (string=? word (second acc)))
```

A list is a poor choice here because it does not hide anything from the client. A client can easily access the password in the list and create a fake account by constructing a new list.

We want to ensure that the client can only create an account through our interface and cannot inspect the password directly.

To solve our problem, we can simply *wrap* a structure around our two-element list.

```
(struct account (lst))

(define (create-account username password)
  (account (list username password)))

(define (get-username acc)
  (first (account-lst acc)))

(define (correct-password? acc word)
  (string=? word (second (account-lst acc))))
```

This may not seem like much of a change, but in Racket it is significantly different.

The definition (`struct account (lst)`) automatically generates the functions `account`, `account?` and `account-lst`.

However, we **did not** `provide` any of those functions. As a result, the client is unable to "peek" inside of an account to view the password.

The client is also unable to "forge" an account. An account can only be created by the `create-account` interface function.

This is an example of achieving **security** through **information hiding**.

In addition, full Racket structures are ***opaque*** by default.

Adding `#:transparent` to a `struct` definition makes the structure "not opaque" or *transparent*.

```
(struct account (lst) #:transparent)
```

The only significant difference is that the fields of a transparent structure can be viewed in output (*e.g.,* in the DrRacket interactions window). This is useful while debugging and testing, but it is not very secure.

Both opaque and transparent structures require field selector functions (*e.g.,* `account-lst`) to obtain field values.

Now that we are using a wrapper structure, it seems more natural to store the username and password as two separate fields in the structure instead of a two-element list.

```
(struct account (uname pword))

(define (create-account username password)
  (account username password))

(define (get-username acc)
  (account-uname acc))

(define (correct-password? acc word)
  (string=? word (account-pword acc)))
```

Once again, we can change the *implementation* without changing the *interface*. This is an example of achieving **flexibility** through **information hiding**.

Data structures & abstract data types

In the previous `account` example, we demonstrated three **implementations** with three different ***data structures***:

- a two-element list
- a `struct` with one field (a two-element list)
- a `struct` with two fields (`uname` and `pword`)

For each *data structure* we knew how the data was "structured".

However, the client doesn't need to know how the data is structured. The client only requires an **abstract** understanding that an `account` stores data (a username and a password).

The `account` module is an implementation of an account **Abstract Data Type (ADT)**.

Formally, an ADT is a mathematical model for storing and accessing data through ***operations***. As mathematical models they transcend any specific computer language or implementation.

However, in practice (and in this course) ADTs are **implemented** as data storage **modules** that only allow access to the data through interface functions (ADT *operations*). The underlying data structure and implementation of an ADT is **hidden** from the client (which provides *flexibility* and *security*).

Data structures *vs.* ADTs

The difference between a *data structure* and an *ADT* is subtle and worth reinforcing.

With a **data structure**, you know how the data is "structured" and you can access the data directly in any manner you desire.

However, with an **ADT** you do not know how the data is structured and you can only access the data through the interface functions (operations) provided by the ADT.

The terminology is especially confusing because the *implementation* of an ADT uses a data structure.

Collection ADTs

The account ADT is not a "typical" ADT because it stores a fixed amount of data and it has limited use.

A ***Collection ADT*** is an ADT designed to store an arbitrary number of items. Collection ADTs have well-defined operations and are useful in many applications.

In CS 135 we were introduced to our first *collection ADT*: a **dictionary**.

In most contexts, when someone refers to an ADT they *implicitly* mean a "collection ADT".

By some definitions, collection ADTs are the *only* type of ADT.

Dictionary (revisited)

The dictionary ADT (also called a *map, associative array, or symbol table*), is a collection of **pairs** of ***keys*** and ***values***. Each *key* is unique and has a corresponding value, but more than one key may have the same value.

Typical dictionary ADT operations:

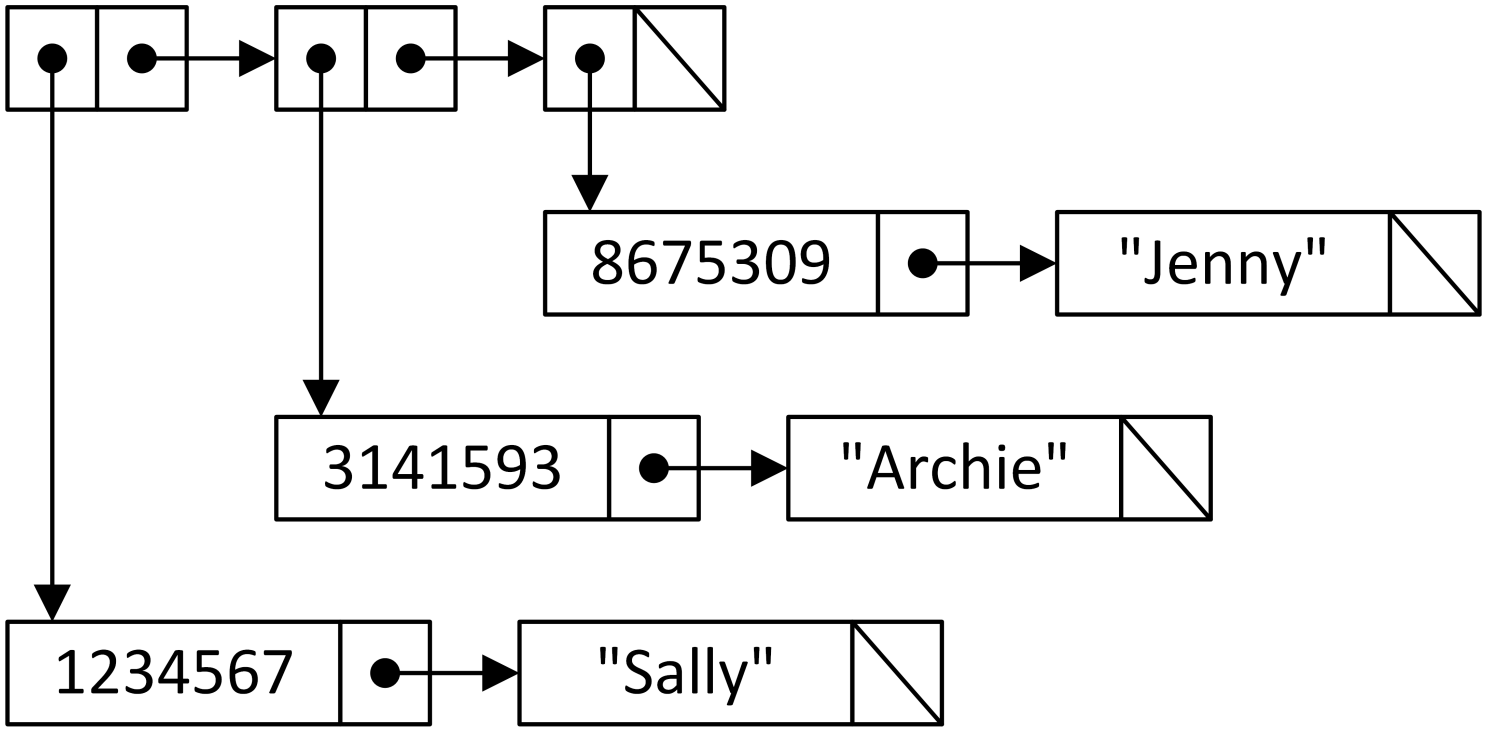
- **lookup:** for a given key, retrieve the corresponding value or "not found"
- **insert:** adds a new key/value pair (or replaces the value of an existing key)
- **remove:** *deletes* a key and its value

example: student numbers

key (student number)	value (student name)
`1234567`	`"Sally"`
`3141593`	`"Archie"`
`8675309`	`"Jenny"`

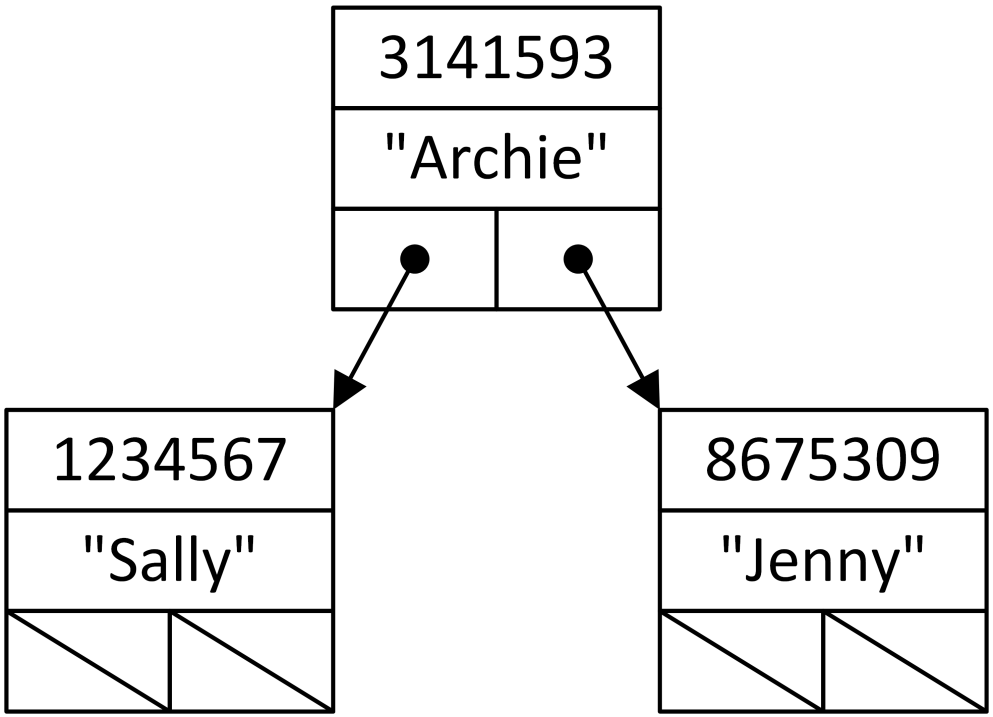
We can implement a dictionary with an ***association list*** *data structure* (a list of key/value pairs with each pair stored as a two-element list).

```
(define al '((1234567 "Sally") (3141593 "Archie")
             (8675309 "Jenny")))
```



Alternatively, we can implement a dictionary with a ***Binary Search Tree (BST)*** *data structure*.

```
(define bst (make-node 3141593 "Archie"
                       (make-node 1234567 "Sally" empty empty)
                       (make-node 8675309 "Jenny" empty empty)))
```



Racket BSTs are briefly reviewed in Appendix A.1.

To *implement* a dictionary, we have a choice:

use an association list, a BST or perhaps something else?

This is a **design decision** that requires us to know the advantages and disadvantages of each choice.

You likely have an intuition that BSTs are “more efficient” than association lists.

In Section 09 we explore what it means to be “more efficient”, and introduce a formal notation to describe the efficiency of an implementation.

More collection ADTs

Three additional collection ADTs that will be explored in this course are:

- stack
- queue
- sequence

Stack ADT

The stack ADT is a collection of items that are “stacked” on top of each other. Items are *pushed* onto the stack and *popped* off of the stack. A stack is known as a LIFO (last in, first out) system. Only the “top” item is accessible.

Stacks are often used in browser histories (“back”) and text editor histories (“undo”).

In Section 05 we will see a very practical use for a stack.

Typical stack ADT operations:

- **push:** adds an item to the top stack
- **pop:** removes the top item from the stack
- **top:** returns the top item on the stack
- **is-empty:** determines if the stack is empty

Stack ADT *vs.* Racket list

The stack ADT is very similar to a Racket list.

The operations: `push/pop/top/is-empty` are closely related to: `cons/rest/first/empty?`.

One significant difference is that with a stack *ADT*, only the top item on the stack is accessible.

With a list *data structure*, any element is accessible (*e.g.,* `second`).

However, we can easily **implement** a Stack ADT by using the Racket list data structure.

```
;; A Stack ADT in Racket (INTERFACE)

(provide create-stack stack-push stack-pop stack-top stack-empty?)

;; (create-stack) creates a new stack
;; create-stack: Void -> Stack

;; (stack-push i s) adds i to the top of stack s
;; stack-push: Any Stack -> Stack

;; (stack-pop s) removes the top item from stack s
;; requires: s is non-empty
;; stack-pop: Stack -> Stack

;; (stack-top s) produces the top item of stack s
;; requires: s is non-empty
;; top: Stack -> Any

;; (stack-empty? s) determines if stack s is empty
;; stack-empty?: Stack -> Bool
```

```
;; A Stack ADT in Racket (IMPLEMENTATION)

(struct stack (lst))

(define (create-stack)
  (stack empty))

(define (stack-push i s)
  (stack (cons i (stack-lst s))))

(define (stack-pop s)
  (stack (rest (stack-lst s))))

(define (stack-top s)
  (first (stack-lst s)))

(define (stack-empty? s)
  (empty? (stack-lst s)))
```

Queue ADT

A queue is like a “lineup”, where new items go to the “back” of the line, and the items are removed from the “front” of the line. While a stack is LIFO, a queue is FIFO (first in, first out).

Typical queue ADT operations:

- **add-back:** adds an item to the end of the queue
- **remove-front:** removes the item at the front of the queue
- **front:** returns the item at the front
- **is-empty:** determines if the queue is empty

Sequence ADT

The sequence ADT is useful when you want to be able to retrieve, insert or delete an item at any position in a sequence of items.

Typical sequence ADT operations:

- **item-at:** returns the item at a given position
- **insert-at:** inserts a new item at a given position
- **remove-at:** removes an item at a given position
- **length:** return the number of items in the sequence

The **insert-at** and **remove-at** operations change the position of items after the insertion/removal point.

Goals of this Section

At the end of this section, you should be able to:

- explain and demonstrate the three core advantages of modular design: abstraction, re-usability and maintainability
- identify two characteristics of a good modular interface: high cohesion and low coupling
- explain and demonstrate information hiding and how it supports both security and flexibility
- explain what a modular interface is, the difference between an interface and an implementation, and the importance of a good interface design.

- use `provide` and `require` to implement modules in Racket
- implement an ADT module in Racket
- produce good interface documentation, including the new documentation changes introduced