CS 320: Principles of Programming Languages

Mark P Jones and Andrew Tolmach
Portland State University

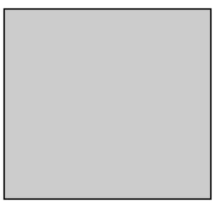
Spring 2019

Week 8: Modules and Abstract Datatypes (ADTs)

How can we construct large programs?

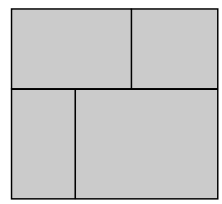
- "Miller's Law" argues that human "working memory" can process only ~7 things at once
- Real-world software systems can have millions of lines of code written by thousands of programmers over dozens of years
- No one person can have a complete, detailed view or understanding of the full system
- We need a **modular** approach to system construction
  - So programmers can work on one module without knowing the internal details of other modules
  - So one module can be updated without breaking other modules

Modular architectures



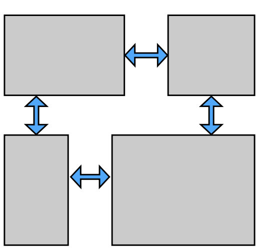
monolithic
program

Modular architectures



divided into
modules

Modular architectures



divided into modules

connected by interfaces

Modules

- **Modules** are units of program code that typically contain definitions of multiple types, constants, variables, functions, etc.
- Modules are useful for
  - namespace control (hiding implementation details)
  - building abstractions (packaging reusable code/functionality)
  - improving build times (via separate compilation)
- Many different designs, with wide variation in features and expressive power. Examples: modules in Haskell or Modula-2; packages in Java; structures/functors in Standard ML; ...
- A good module system facilitates modular software construction and software reuse

Module interfaces

*not necesarily a Java `interface`

- An **interface*** describes how two modules interact
- It describes the **services** that each module exports to its **client** modules
- Representation of interfaces varies widely among languages
- The formal interface description typically includes

  the name of each public function exported via the interface

  the **signature** of each function (i.e., its argument types and return types, assuming the language is statically typed)
- Informally (in comments) we also might expect/hope for:

  a description of what each function does

  information on its efficiency, resource use, etc.

Modularity supports large developments

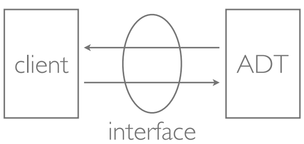
- To use a service, we need to know *what* it does ...
  - but we do not need to know *how* its operations are implemented or its data is represented
- A service can change its choice of algorithm or data representation…
- … as long as the interface signatures do not change, clients do not need to be re-written or (maybe) even re-compiled
  - (although performance and resource use may be impacted)
- One important way to modularize code is to use ADTs ...

Abstract datatypes

- An **abstract datatype** (ADT) provides a well-defined interface to:

  a named type

  an associated group of operations



- Ideally, the interface should specify behavior precisely so that the clients and the implementation of an ADT can be written, tested, and maintained independently
- In practice, we often get types and informal descriptions, but other details (even as simple as algorithmic complexity) are often omitted, which can make an ADT harder to use

Example

- The classic example (in hypothetical syntax):

```
interface Stack {
  Stack newStack();
  bool isEmpty(Stack);
  void push(Stack, int);
  int top(Stack) throws EmptyStackException;
  void pop(Stack) throws EmptyStackException;
}
```

- Now a client can use a stack without knowing about its implementation:

```
Stack s = newStack();
push(s, 1);
push(s, 2);
push(s, 3);
pop(s);
print 40 + top(s);
```

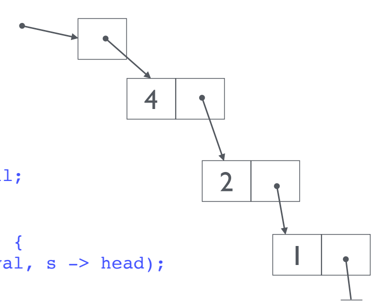

List based implementation

```
implement Stack as HeadPtr {
  typedef Head* HeadPtr;
  typedef struct { IntList* head; } Head;

  Stack newStack() {
    HeadPtr s = new Head();
    s -> head = null;
    return s;
  }

  bool isEmpty(Stack s) {
    return (s -> head) == null;
  }

  void push(Stack s, int val) {
    s -> head = new IntList(val, s -> head);
  }
  ...
}
```

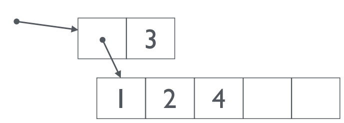


Array based implementation

```
implement Stack as HeadPtr {
  typedef Head* HeadPtr;
  typedef struct { int[] vals, int count; } Head;

  Stack newStack() {
    HeadPtr s = new Head();
    s->vals   = new int[0];
    s->count  = 0;
  }

  void push(Stack s, int val) {
    if (s->count >= length(s->vals)) {
      allocate a bigger values array, copy old vals into new,
      delete old array, set s->vals to point to new array
    }
    s->vals[s->count++] = val;
  }
  ...
}
```



Encapsulation

- An ADT **encapsulates** internal state so that it is "hidden" and can only be accessed from code in its implementation
- External code cannot access or overwrite an arbitrary element of the stack
- System invariants are preserved and protected by the ADT
  - For example, there is no way to create an array-based stack in which the count exceeds the length of the array, or in which the count is negative

ADTs in practice, Java Style

- In Java, ADTs can be implemented using private state:

```
class Stack {
    private int count = 0;                    private state
    private int[] vals = new int[0];
    private Stack() { }                       private methods

    public newStack() { return new Stack(); }

    public push(int x) {                      public methods
        if (count>=vals.length) {
            ...
        }
        vals[count++] = x;
    }
    ...
}
```


ADTs in practice, Haskell style

- In Haskell, ADTs can be implemented using modules:

```
module Stack(Stack, newStack, push, pop, …) where

data Stack a = MkStack [a]

newStack :: Stack a
newStack  = MkStack []

push               :: a -> Stack a -> Stack a
push x (MkStack xs) = MkStack (x:xs)

…
```

limits exports, hiding implementation

The MkStack constructor is not exported to clients, so they cannot build or examine stacks directly

ADTs and static analysis

- Static analysis plays a major role in the implementation of ADTS:

  enforcing typing rules
  restricting accessibility (e.g., public, private, protected, ...)

- In a sense, ADTs provide a form of user-defined static analysis by restricting the ways in which ADT values can be used

Problems with ADT interfaces

- It can be difficult to define an interface in precise terms:

```
abstype Stack {
  Stack newStack();
  bool isEmpty(Stack);
  void push(Stack, int);
  int top(Stack) throws EmptyStackException;
  void pop(Stack) throws EmptyStackException;
}
```

- This interface doesn't answer questions like the following:

  1) is there a limit on stack size?
  2) how long does it take to push an element on to the stack?
  3) what conditions trigger an EmptyStackException exception?
  4) if we push(s, x) and then call top(), what value do we get?