Notes on Data Abstraction: Chapters 1-2

H. Conrad Cunningham

10 June 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of June 2022 is a recent version of Firefox from Mozilla.

1 Data Abstraction Concepts

1.1 Chapter Introduction

TODO: What are the goals of document/chapter?

1.2 What is Abstraction?

As computing scientists and computer programmers, we should remember the maxim:

> *Simplicity is good; complexity is bad.*

The most effective weapon that we have in the fight against complexity is *abstraction*. What is abstraction?

Abstraction is *concentrating on the essentials and ignoring the details.*

Sometimes abstraction is described as *remembering the "what" and ignoring the "how"*.

Large complex problems can only be made understandable by decomposing them into subproblems. Ideally, we should be able to solve each subproblem independently and then compose their solutions into a solution to the larger problem.

In programming, the subproblem solution is often represented by an abstraction expressed in a programming notation. From the outside, each abstraction should be simple and easy for programmers to use correctly. The programmers should only need to know the abstraction's *interface* (i.e., some small number of assumptions necessary to use the abstraction correctly).

TODO: I am unsure about the use of the term "abstraction" in the above paragraph here and in ELIFP chapter 2.

1.2.1 Kinds of abstraction

Two kinds of abstraction are of interest to computing scientists: *procedural abstraction* and *data abstraction*.

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

In procedural abstraction, programmers focus primarily on the actions to be carried out and secondarily on the data to be processed.

For example, in the top-down design of a sequential algorithm, a programmer first identifies a sequence of actions to solve the problem without being overly concerned about how each action will be carried out.

If an action is simple, the programmer can code it directly using a sequence of programming language statements.

If an action is complex, the programmer can abstract the action into a subprogram (e.g., a procedure or function) in the programming language. The programmer must define the subprogram's name, parameters, return value, effects, and assumptions—that is, define its interface. The programmer subsequently develops the subprogram using the same top-down design approach.

In data abstraction, programmers primarily focus on the problem's data and secondarily on its actions. Programmers first identify the key data representations and develop the programs around those and the operations needed to create and update them.

TODO: Do we need to reintroduce the idea of module into the text above?

1.2.2 Procedures and functions

Generally we make the following distinctions among subprograms:

- A *procedure* is (in its pure form) a subprogram that takes zero or more arguments but does not return a value. It is executed for its effects, such as changing values in a data structure within the program, modifying its reference or value-result arguments, or causing some effect outside the program (e.g., displaying text on the screen or reading from a file).
- A *function* is (in its pure form) a subprogram that takes zero or more arguments and returns a value but that does not have other effects.
- A *method* is a procedure or function often associated with an object or class in an object-oriented program. Some object-oriented languages use the metaphor of message-passing. A method is the feature of an object that receives a message. In an implementation, a method is typically a procedure or function associated with the (receiver) object; the object may be an *implicit parameter* of the method.

Of course, the features of various programming languages and usual practices for their use may not follow the above pure distinctions. For example, a language may not distinguish between procedures and functions. One term or another may be used for all subprograms. Procedures may return values. Functions may have side effects. Functions may return multiple values. The same subprogram can sometimes be called either as a function or procedure.

Nevertheless, it is good practice to maintain the distinction between functions and procedures for most cases in software design and programming.

In Haskell, the primary unit of procedural abstraction is the pure function. Their definitions may be nested within other functions. Haskell also groups functions and other declarations into a program unit called a `module`. A `module` explicitly exports selected functions and keep others hidden.

In Java, the primary unit of procedural abstraction is the method, which may be a procedure, a pure function, or an impure (side-effecting) function. A method must be part of a Java `class`, which is, in turn, part of a Java `package`.

Scala combines characteristics of Java and Haskell. It has Java-like methods, which can be nested inside other methods. Scala also has `class`, `object`, `trait`, and `package` constructs that include procedural abstractions.

1.3 Concrete Data Structures

In most languages (e.g., C), data structures are visible. A programmer can define custom data types, yet their structure and values are known to other parts of the program. These are *concrete data structures*.

As an example, consider a collection of records about the employees of a company. Suppose we store these records in a global C array. The array and all its elements are visible to all parts of the program. Any statement in the program can directly access and modify the elements of the array.

The use of concrete data structures is convenient, but it does not scale well and it is not robust with respect to change. As a program gets large, keeping track of the design details of many concrete data structures becomes very difficult. Also, any change in the design or implementation of a concrete data structures may require change to all code that uses it.

1.4 Abstract Data Structures

TODO: The following uses the term module in a general manner, so it may be necessary to introduce the concept of module above. (I had some changes above to the current version—by bringing some text back from ELIFP Chapter 2.

An *abstract data structure* is a module consisting of data and operations. The data are hidden within the module and can only be accessed by means of the operations. The data structure is called *abstract* because its name and its interface are known, but not its implementation. The operations are explicitly given; the values are only defined implicitly by means of the operations.

An abstract data structure supports *information hiding*. Its implementation is hidden behind an interface that remains unchanged, even if the implementation changes. The implementation detail of the module is a design decision that is kept as a *secret* from the other modules.

The concept of *encapsulation* is related to the concept of information hiding. The data and the operations that manipulate the data are all combined in one place. That is, they are encapsulated within a module.

An abstract data structure has a *state* that can be manipulated by the operations. The state is a value, or collection of information, held by the abstract data structure.

As an example, again consider the collection of records about the employees of a company. Suppose we impose a discipline on our program, only allowing the collection of records to be accessed through a small group of procedures (and functions). Inside this group of procedures, the array of records can be manipulated directly. However, all other parts of the program must use one of the procedures in the group to manipulate the records in the collection. The fact that the collection is implemented with an array is (according to the discipline we imposed) hidden behind the interface provided by the group of procedures. It is a secret of the module providing the procedures.

Now suppose we wish to modify our program and change the implementation from an array to a linked list or maybe to move the collection to a disk file. By approaching the design of the collection as an abstract data structure, we have limited the parts of the program that must be changed to the small group of procedures that used the array directly; other parts of the program are not affected.

As another example of an abstract data structure, consider a stack. We provide operations like `push`, `pop`, and `empty` to allow a user of the stack to access and manipulate it. Except for the code implementing these operations, we disallow direct access to the concrete data structure that implements the stack. The implementation might use an array, a linked list, or some other concrete data structure; the actual implementation is "hidden" from the user of the stack.

We, of course, can use the available features of a particular programming language (e.g., module, package, class) to hide the implementation details of the data structure and only expose the access procedures.

1.5 Abstract Data Types

There is only one instance of an abstract data structure. Often we need to create multiple instances of an abstract data structure. For example, we might need to have a collection of employee records for each different department within a large company.

We need to go a step beyond the abstract data structure and define an *abstract data type* (ADT).

What do we mean by *type*?

Type: a category of entities sharing common characteristics

Consider the built-in type `int` in C. By declaring a C variable to be of type `int`, we are specifying that the variable has the characteristics of that type:

1. a value (state) drawn from some set (domain) of possible values—in the case of `int`, a subset of the mathematical set of integers
2. a set of operations that can be applied to those values—in the case of `int`, addition, multiplication, comparison for equality, etc.

Suppose we declare a C variable to have type `int`. By that declaration, we are creating a container in the program's memory that, at any point in time, holds a single value drawn from the `int` domain. The contents of this container can be operated upon by the `int` operations. In a program, we can declare several `int` variables: each variable may have a different value, yet all of them have the same set of operations.

In the definition of a *concrete* data type, the values are the most prominent features. The values and their representations are explicitly prescribed; the operations on the values are often left implicit.

The opposite is the case in the definition of an *abstract* data type. The operations are explicitly prescribed; the values are defined implicitly in terms of the operations. A number of representations of the values may be possible.

Conceptually, an abstract data type is a set of entities whose logical behavior is defined by a domain of values and a set of operations on that domain. In the terminology we used above, an ADT is set of abstract data structures all of whom have the same domain of possible states and have the same set of operations.

We will refer to a particular abstract data structure from an ADT as an *instance* of the ADT.

The implementation of an ADT in a language like C is similar to that discussed above for abstract data structures. In addition to providing operations to access and manipulate the data, we need to provide operations to create and destroy instances of the ADT. All operations (except create) must have as a parameter an identifier (e.g., a pointer) for the particular instance to be operated upon.

While languages like C do not directly support ADTs, the `class` construct provides a direct way to define ADTs in languages like C++, Java, and Scala.

1.6 Defining ADTs

The behavior of an ADT is defined by a set of operations that can be applied to an *instance* of the ADT.

Each operation of an ADT can have inputs (i.e., parameters) and outputs (i.e., results). The collection of information about the names of the operations and their inputs and outputs is the *interface* of the ADT.

To specify an ADT, we need to give:

1. the *name* of the ADT
2. the *sets* (or domains) upon which the ADT is built. These include the type being defined and the auxiliary types (e.g., primitive data types and other ADTs) used as parameters or return values of the operations.
3. the *signatures* (syntax or structure) of the operations
    - name

    - input sets (i.e., the types, number, and order of the parameters)
    - output set (i.e., the type of the return value)
4. the *semantics* (or meaning) of the operations

There are two primary approaches for specifying the semantics of the operations:

- The *axiomatic* (or *algebraic*) approach gives a set of logical rules (properties or axioms) that relate the operations to one another. The meanings of the operations are defined implicitly in terms of each other.
- The *constructive* (or *abstract model*) approach describes the meaning of the operations explicitly in terms of operations on other abstract data types. The underlying *model* may be any well-defined mathematical model or a previously defined ADT.

In some ways, the axiomatic approach is the more elegant of the two approaches. It is based in the well-established mathematical fields of abstract algebra and category theory. Furthermore, it defines the new ADT independently of other ADTs. To understand the definition of the new ADT it is only necessary to understand its axioms, not the semantics of a model.

However, in practice, the axiomatic approach to specification becomes very difficult to apply in complex situations. The constructive approach, which builds a new ADT from existing ADTs, is the more useful methodology for most practical software development situations.

To illustrate both approaches, let us look at a well-known ADT that we studied in the introductory data structures course, the stack.

1.7 Axiomatic Specification of an Unbounded Stack ADT

In this section we give an axiomatic specification of an unbounded stack ADT. By unbounded, we mean that there is no maximum capacity for the number of items that can be pushed onto an instance of a stack.

Remember that an ADT specification consists of the name, sets, signatures, and semantics.

1.7.1 Name

`Stack` (of `Item`)

In this specification, we are defining an ADT named `Stack`. The parameter `Item` represents the arbitrary unspecified type for the entities stored in the stack. `Item` is a formal *generic parameter* of the ADT specification. `Stack` is itself a generic ADT; a different ADT is specified for each possible *generic argument* that can be substituted for `Item`.

1.7.2 Sets

The sets (domains) involved in the `Stack` ADT are the following:

`Stack:` the set of all stack instances
(This is the set we are defining with the ADT.)
`Item:` the set of all items that can appear in a stack instance
`boolean:` the primitive Boolean type `{ False, True }`

1.7.3 Signatures

To specify the signatures for the operations, we use the notation for mathematical functions. By a tuple like `(Stack, Item)`, we mean the Cartesian product of sets `Stack` and `Item`, that is, the set of ordered pairs where the first component is from `Stack` and the second is from `Item`. The set to the right of the `->` is the return type of the function.

We categorize the operations into one of four groups depending upon their functionality:

- A *constructor* (sometimes called a creator, factory, or producer function) constructs and initializes an instance of the ADT.
- A *mutator* (sometimes called a modifier, command, or "setter" function) returns the instance with its state changed.
- An *accessor* (sometimes called an observer, query, or "getter" function) returns information from the state of an instance without changing the state.
- A *destructor* destroys an instance of the ADT.

We will normally list the operations in that order.

For now, we assume that a mutator returns a distinct new instance of the ADT with a state that is a modified version of the original instance's state. That is, we are taking an applicative (or functional or referentially transparent) approach to ADT specifications.

Technically speaking, a destructor is not an operation of the ADT. We can represent the other types of operations as functions on the sets in the specification. However, we cannot define a destructor in that way. But destructors are of pragmatic importance in the implementation of ADTs, particularly in languages that do not have automatic storage reclamation (i.e., garbage collection).

The signatures of the `Stack` ADT operations are as follows.

1.7.3.1 Constructors `create:  -> Stack`

1.7.3.2 Mutators `push: (Stack, Item) -> Stack`

`pop: Stack -> Stack`

1.7.3.3 Accessors `top: Stack -> Item`

`empty: Stack -> boolean`

1.7.3.4 Destructors

```
destroy: Stack ->
```

The operation `pop` may not be the same as the "pop" operation you learned in a data structures class. The traditional "pop" both removes the top element from the stack and returns it. In this ADT, we have separated out the "return top" functionality into accessor operation `top` and left operation `pop` as a pure mutator operation that returns the modified stack.

The separation of the traditional "pop" into two functions has two advantages:

1. It results in an elegant, applicative stack specification whose operations fit cleanly into the mutator/accessor categorization.
2. It results in a simpler, cleaner abstraction in which the set of operations is "atomic". No operation in the ADT's interface can be decomposed into other operations also in the interface.

Also note that operation `destroy` does not return a value. As we pointed out above, the `destroy` operation is not really a part of the formal ADT specification.

1.7.4 Semantics (axiomatic approach)

We can specify the semantics of the `Stack` ADT with the following axioms. Each axiom must hold for all instances `s` of type `Stack` and all entities `x` of type `Item`.

1. `top(push(s,x)) = x`
2. `pop(push(s,x)) = s`
3. `empty(create()) = True`
4. `empty(push(s,x)) = False`

The axioms are logical assertions that must always be true. Thus we can write Axioms 3 and 4 more simply as:

3. `empty(create())`
4. `not empty(push(s,x))`

The first two axioms express the last-in-first-out (LIFO) property of stacks. Axiom 1 tells us that the top element of the stack is the last element pushed. Axiom 2 tells us that removal of the top element returns the stack to the state it had before the last push.

Moreover, axioms 1 and 2 specify the LIFO property of stacks in purely mathematical terms; there was no need to use the properties of any representation or use any time-based (i.e., imperative) reasoning.

The last two axioms define when a stack is empty and when it not. Axiom 3 tells us that a newly created stack is empty. Axiom 4 tells us that pushing an entity on a stack results in a nonempty stack.

But what about the sequences of operations `top(create())` and `pop(create())`?

Clearly we do not want to allow either `top` or `pop` to be applied to an empty stack. That is, `top` and `pop` are undefined when their arguments are empty stacks.

Functions may be either total or partial.

- A *total* function `A -> B` is defined for all elements of `A`.

  For example, the multiplication operation on the set of real numbers `R` is a total function `(R,R) -> R`.

- A *partial* function `A -> B` is undefined for one or more elements of `A`.

  For example, the division operation on the set of real numbers `R` is a partial function because it is undefined when the divisor is `0`.

In software development (and, hence, in specification of ADTs), partial functions are common. To avoid errors in execution of such functions, we need to specify the actual domain of the partial functions precisely.

In an axiomatic specification of an ADT, we restrict operations to their domains by using preconditions. The *precondition* of an operation is a logical assertion that specifies the assumptions about and the restrictions upon the values of the arguments of the operation.

If the precondition of an operation is false, then the operation cannot be safely applied. If any operation is called with its precondition false, then the program is *incorrect*.

In the axiomatic specification of the stack, we introduce two preconditions as follows.

Precondition of `pop(Stack S)`: `not empty(S)`
Precondition of `top(Stack S)` `not empty(S)`

Note that we have not given the semantics of the destructor operation `destroy`. This operation cannot be handled in the simple framework we have established.

Operation `destroy` is really an operation on the "environment" that contains the stack. By introducing, the "environment" explicitly into our specification, we could specify its behavior more precisely. Of course, the semantics of `create` would also need to be extended to modify the environment and the other operations would likely require preconditions to ensure that the stack has been created in the environment.

Another simplification that we have made in this ADT specification is that we did not impose a bound on the capacity of the stack instance. We could specify this, but it would also complicate the axioms the specification.

1.8 Constructive Specification of a Bounded Stack ADT

In this section, we give a constructive specification of a bounded stack ADT. By bounded, we mean that there is a maximum capacity for the number of items that can be pushed onto an instance of a stack.

1.8.1 Name

`StackB` (of `Item`)

1.8.2 Sets

In this specification of bounded stacks, we have one additional set involved, the set of integers.

`StackB`: the set of all stack instances
`Item`: set of all items that can appear in a stack instance
`boolean`: the primitive Boolean type
`integer`: the primitive integer type `{ ..., -2, -1, 0, 1, 2, ... }`

1.8.3 Signatures

In this specification of unbounded stacks, we define the `create` operation to take the maximum capacity as its parameter.

1.8.3.1 Constructors `create: integer -> StackB`

1.8.3.2 Mutators `push: (StackB, Item) -> StackB`

`pop: StackB -> StackB`

In this specification, we add operation `full` to detect whether or not the stack instance has reached its maximum capacity.

1.8.3.3 Accessors `top: StackB -> Item`

`empty: StackB -> boolean`

`full: StackB -> boolean`

1.8.3.4 Destructors `destroy: StackB ->`

1.8.4 Semantics (constructive approach)

In the constructive approach, we give the semantics of each operation by associating both a precondition and a postcondition with the operation.

As before, the *precondition* is a logical assertion that specifies the required characteristics of the values of the arguments.

A *postcondition* is a logical assertion that specifies the characteristics of the result computed by the operation with respect to the values of the arguments.

In the specification in this subsection, we are a bit informal about the nature of the underlying model. Although the presentation here is informal, we try to be precise in the statement of the pre- and postconditions.

Note: We can formalize the model using an ordered pair of type `(integer max, sequence stkseq)`, in which `max` is the upper bound on the stack size and `stkseq` is a sequence that represents the current sequence elements of elements in the stack. This, more formal alternative, is presented in the next subsection.

1.8.4.1 Constructor `create(integer size) -> StackB S'`

- **Precondition:** `size >= 0`
- **Postcondition:** `S'` is a valid new instance of `StackB` &&
  `S'` has the capacity to store `size` items &&
  `empty(S')`

1.8.4.2 Mutators `push(StackB S, Item I) -> StackB S'`

- **Precondition:** `S` is a valid `StackB` instance &&
  `not full(S)`
- **Postcondition:** `S'` is a valid `StackB` instance &&
  `S' = S` with `I` added as the new top.

`pop(StackB S) -> StackB S'`

- **Precondition:** `S` is a valid `StackB` instance &&
  `not empty(S)`
- **Postcondition:** `S'` is a valid `StackB` instance &&
  `S' = S` with the top item deleted

1.8.4.3 Accessors `top(StackB S) -> Item I`

- **Precondition:** `S` is a valid `StackB` instance &&
  `not empty(S)`
- **Postcondition:** `I =` the top item on `S`
  (`S` is not modified by this operation.)

`empty(StackB S) -> boolean e`

- **Precondition:** `S` is a valid `StackB` instance

- **Postcondition:** `e` is `true` if and only if `S` contains no elements (i.e., is empty)
  (`S` is not modified by this operation.)

`full(StackB S) -> boolean f`

- **Precondition:** `S` is a valid `StackB` instance
- **Postcondition:** `f` is `true` if and only if `S` contains no space for additional items (i.e., is full)
  (`S` is not modified by this operation.)

1.8.4.4 Destructor `destroy(StackB S) ->`

- **Precondition:** `S` is a valid `StackB` instance
- **Postcondition:** `StackB S` no longer exists

Note that each operation except the constructor (`create`) has a `StackB` instance as an input; the constructor and each of the mutators also has a `StackB` instance as an output. This parameter identifies the particular instance that the operation is manipulating.

Also note that all of these `StackB` instances are required to be "valid" in all preconditions and postconditions, except the precondition of the constructor and the postcondition of the destructor. By valid we mean that the state of the instance is within the acceptable domain of values; it has not become corrupted or inconsistent. What is specifically mean by "valid" will differ from one implementation of a stack to another.

Suppose we implement the mutator operations as imperative commands rather then applicative functions. That is, we implement mutators so that they directly modify the state of an instance instead of returning a modified copy. (`S` and `S'` are implemented as different states of the same physical instance.)

Then, in some sense, the above "validity" property is *invariant* for an instance of the ADT; the constructor makes the property true, all mutators and accessors preserve its truth, and the destructor makes it false.

An invariant property must hold between operations on the instance; it might not hold during the execution of an operation. (For this discussion, we assume that only one thread has access to the ADT implementation.)

Aside: An invariant on an ADT instance is similar in concept to an invariant for a while-loop. A loop invariant holds before and after each execution of the loop.

As a convenience in specification we will sometimes state the invariants of the ADT separately from the pre- and postconditions of the methods. We sometimes will divide the invariants into two groups.

interface invariants: invariants stated in terms of publicly accessible features and abstract properties of the ADT instance.

***implementation (representation) invariants*:** detailed invariants giving the required relationships among the internal data fields of the implementation.

The interface invariants are part of the public interface of the ADT. They only deal with the state of an instance in terms of the abstract model for the ADT.

The implementation invariants are part of the hidden state of an instance; in some cases, they define the meaning of the abstract properties stated in the interface invariants in terms of hidden values in the implementation.

1.8.5 More formal semantics for bounded stack

Let the bounded stack `StackB` be represented by an ordered pair of type `(integer max, sequence stkseq)`, in which `max` is the upper bound on the stack size and `stkseq` is a sequence that represents the current sequence elements of elements in the stack.

1.8.5.1 Constructor `create(integer size) -> StackB S'`

- **Precondition:** `size >= 0`
- **Postcondition:** `S' == (size,[])`

  Here `[]` represents an empty sequence. The value of a variable occurring in the postcondition is the same as that variable's value in the precondition.

1.8.5.2 Mutators `push(StackB S, Item I) -> StackB S'`

- **Precondition:** `S == (m,ss) && m >= 0 && length(ss) < m`
- **Postcondition:** `S' == (m,[I]++ss)`

  Above `++` denotes the concatenation of its left and right operand sequences. The result sequence has all the values from the left operand sequence, in the same order, followed by all the values from the right operand sequence, in the same order. Also the notation `[I]` represents a sequence consisting a single element with the value `I`.

`pop(StackB S) -> StackB S'`

- **Precondition:** `S == (m,ss) && m >= 0 && length(ss) > 0`
- **Postcondition:** `S' == (m,tail(ss))`

  Above `tail` is a function that returns the sequence remaining after removing the first element of its nonempty sequence argument. Similarly, the function `head` (used below) returns the first element of its nonempty sequence argument.

1.8.5.3 Accessors `top(StackB S) -> Item I`

- **Precondition:** `S == (m,ss) && m >= 0 && length(ss) > 0`
- **Postcondition:** `I = head(ss) && S' == S`

`empty(StackB S) -> boolean e`

- **Precondition:** `S == (m,ss) && m >= 0 && length(ss) <= m`
- **Postcondition:** `e == (length(ss) == 0)  && S' == S`

`full(StackB S) -> boolean f`

- **Precondition:** `S == (m,ss) && m >= 0  && length(ss) <= m`
- **Postcondition:** `f == (length(ss) == m) && S' == S`

1.8.5.4 Destructor `destroy(StackB S) ->`

- **Precondition:** `S == (m,ss) && m >= 0  && length(ss)   <= m`
- **Postcondition:** `StackB S` no longer exists

Using this abstract model, we can state an interface invariant:

> For a `StackB S`, there exists an integer `m` and sequence of `Item` elements `l` such that `S == (m,ss) && m >= 0 && length(ss) <= m`

For discussion of implementing ADTs as Java classes, see the chapter Data Abstraction in Java. A Java implementation of the StackB ADT appears in that chaper.

1.9 Date (Day) ADT

Consider an ADT for storing and manipulating calendar dates. We call the ADT `Day` to avoid confusion with the `Date` class in the Java API. This ADT is based on the `Day` class defined in Chapter 4 of the Horstmann and Cornell [10].

Logically, a calendar date consists of three pieces of information: a *year* designator, a *month* designator, and a *day* of the month designator. A secondary piece of information is the day of the week. In this ADT interface definition, we use integers (e.g., Java `int`) to designate these pieces of information.

Caveat: The discussion of Java in these notes does not use generic type parameters.

1.9.1 Constructor

`create(integer y, integer m, integer d) -> Day D'`

- **Precondition:** `y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month m` &&
  `(y,m,d)` does not fall in the gap formed by the change to the modern (Gregorian) calendar
- **Postcondition:** `D'` is a valid new instance of `Day` with year `y`, month `m`, and day `d`

1.9.2 Mutators

`setDay(Day D, integer y, integer m, integer d) -> Day D'`

- **Precondition:** `D` is a valid instance of `Day` && `y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month` &&
  `(y,m,d)` does not fall in the gap formed by the change to the modern (Gregorian) calendar
- **Postcondition:** `D'` is a valid instance of `Day` &&
  `D'= D` except with year `y`, month `m`, and day `d`

  Question: Should we include `setDay`, `setMonth`, and `setYear` operations? What problems might arise?

`advance(Day D, integer n) -> Day D'`

- **Precondition:** `D` is a valid instance of `Day`
- **Postcondition:** `D'` is a valid instance of `Day` &&
  `D' = D` with the date moved `n` days later (Negative `n` moves to an earlier date.)

1.9.3 Accessors

`getDay(Day D) -> integer d`

- **Precondition:** `D` is a valid instance of `Day`
- **Postcondition:** `d` is day of the month from `D`, where `1 <= d <= #days in month getMonth(D)`
  (`D` is unchanged.)

`getMonth(Day D) -> integer m`

- **Precondition:** `D` is a valid instance of `Day`
- **Postcondition:** `m` is the month from `D`, where `1 <= m <= 12`
  (`D` is unchanged.)

`getYear(Day D) -> integer y`

- Precondition: : `D` is a valid instance of `Day`
- **Postcondition:** `y` is the year from `D`, where `y != 0`
  (`D` is unchanged.)

`getWeekday(Day D) -> integer wd`

- **Precondition:** `D` is a valid instance of `Day`
- **Postcondition:** `wd` is the day of the week upon which `D` falls: 0 = Sunday, 1 = Monday, . . . , 6 = Saturday
  (`D` is unchanged.)

`equals(Day D, Day D1) -> boolean eq`

- **Precondition:** `D` and `D'` are valid instances of `Day`
- **Postcondition:** `eq` is true if and only if `D` and `D'` denote the same calendar date
  (`D` and `D'` are unchanged.)

`daysBetween(Day D, Day D1) -> integer d`

- **Precondition:** `D` and `D'` are valid instances of `Day`
- **Postcondition:** `d` is the number of calendar days from `D1` to `D`, i.e., `equals(D,advance(D1,d))` would be true
  (`D` is unchanged.)

`toString(Day D) -> String s`

- **Precondition:** `D` is a valid instance of `Day`
- **Postcondition:** `s` is the date `D` expressed in the format "Day[`getYear(D)`,`getMonth(D)`,`getDay(D)`]".
  (`D` is unchanged.)

Note: This method is a "standard" method that should be defined for most Java classes so that they fit well into the Java language framework.

1.9.4 Destructor

`destroy(Day D) ->`

- Precondition: `D` is a valid instance of `Day`
- Postcondition: `D` no longer exists

A Java implementation of the Day ADT appears in the supplementary notes.

1.10 Client-Supplier Relationship

The design and implementation of ADTs (i.e., classes) must be approached from two points of view simultaneously:

supplier the developers of the ADT — the providers of the services
client the users of the ADT — the users of the services (e.g., the designers of other ADTs)

The *client-supplier relationship* is as represented in the following diagram:

```
 ----------------             ----------------
|                |           |                |
|     Client     |===USES===>|    Supplier    |
|________________|           |________________|

   (ADT user)                     (ADT)
```

TODO: Replace the above diagram with a better graphic and make it a Figure.

The supplier's concerns include:

- efficient and reliable algorithms and data structures,
- convenient implementation,
- easy maintenance.

The clients' concerns include:

- accomplishing their own tasks,
- using the supplier ADT without effort to understand its internal details,
- having a sufficient, but not overwhelming, set of operations.

As we have noted previously, the *interface* of an ADT is the set of features (i.e., public operations) provided by a supplier to clients.

A precise description of a supplier's interface forms a *contract* between clients and supplier.

The client-supplier contract:

1. gives the responsibilities of the client. These are the conditions under which the supplier must deliver results — when the *preconditions* of the operations are satisfied (i.e., the operations are called correctly).

2. gives the responsibilities of the supplier. These are the benefits the supplier must deliver — make the *postconditions* hold at the end of the operation (i.e., the operations deliver the correct results).

The contract

- protects the client by specifying how much must be done by the supplier.

- protects the supplier by specifying how little is acceptable to the client.

If we are both the clients and suppliers in a design situation, we should consciously attempt to separate the two different areas of concern, switching back and forth between our supplier and client "hats".

1.11 Design Criteria for ADT Interfaces

We can use the following design criteria for evaluating ADT interfaces. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

In object-oriented languages, these criteria also apply to class interfaces.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The ADT should describe a single abstraction.
- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the ADT (class).
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the ADT (class).
- **Consistency:** Provide a set of operations that are internally consistent in
  - naming convention (e.g., in use of prefixes like “set” or “get”, in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g., order and type of arguments),
  - behavior (i.e., make operations work similarly).

  Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.
- **Reusability:** Do not customize ADTs (classes) to specific clients, but make them general enough to be reusable in other contexts.
- **Robustness with respect to modifications:** Design the interface of an ADT (class) so that it remains stable even if the implementation of the ADT changes.
- **Convenience:** Where appropriate, provide additional operations (e.g., beyond the complete primitive set) for the convenience of users of the ADT (class). Add convenience operations only for frequently used combinations after careful study.

1.12 What Next?

TODO

1.13 Exercises

TODO

1.14 Acknowledgements

In Spring 2017 I adapted these lecture notes from my previous notes on this topic. The material here is based on my study of a variety of sources (e.g., Bird and Wadler [1], Dale [7], Gries [8]; Horstmann [9,10], Liskov [11], Meyer [12], Mossenbock [13], Parnas [16], and Thomas [18]).

I wrote the first version of these lecture notes to use in the first Java-based version of CSci 211 (then titled File Systems) during Fall 1996. I revised the notes incrementally over the next decade for use in my Java-based courses on object-orientation and software architecture. I partially revised the notes for use in my Scala-based classes beginning in Fall 2008.

In Fall 2013 I updated these notes to better support my courses that used non-JVM languages such as Lua, Elixir, and Haskell. I moved the extensive Java-based content to a separate document and developed separate case studies for the other languages.

In Summer 2017, I adapted the notes to use Pandoc. I continued to revise the structure and text in minor ways in 2017, 2018, and 2019.

I incorporated quite a bit of this material into Chapters 2, 6, and 7 of the 2018 draft of the textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

TODO: 2022 updates

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

1.15 Terms and Concepts

TODO

2 Data Abstraction in Java

2.1 Chapter Introduction

This chapter is a Java-specific supplement to the chapter *Data Abstraction Concepts* [3]. However, most of the concepts apply to other object-oriented languages (e.g., Scala [15,17]. The discussion here is meant to be read in conjunction with the *Data Abstraction Concepts* chapter.

This chapter discusses use of Java classes to implement abstract data types (ADTs). It seeks to use good object-oriented programming practices, but it does not cover the principles and practices of object-oriented programming fully. For more information on object orientation, see my the chapters Object-Oriented Software Development [6] and Object-Based Paradigms [5] (a chapter in the ELIFP textbook [4, Ch. 3]).

Caveat: I wrote this chapter originally in the late 1990s. It should be updated to use Java generics (and possibly other newer Java features such as default methods in interfaces and lambda expressions). However, the basic principles used here are still relevant to contemporary Java programming.

TODO: Update chapter to use generics and possibly other newer Java features as needed.

2.2 Java as an Object-Oriented Language

TODO: Make sure this new section and the older sections that follow use consistent terminology and flow smoothly.

According to the concepts and terminology used in the Object-Based Paradigms [5] chapter of the ELIFP textbook [4, Ch. 3], Java is an object-oriented language. Its object model includes:

- objects

  Java objects are instances of classes.

- classes

  Java classes are declared with the keyword `class`.

- inheritance

  A Java `class` inherits from (i.e., `extends`) one other Java class—the builtin, top-level class `Object` if no class is explicitly specified.

  In addition to the single inheritance from a class, a Java class `implements` zero or more interfaces. A Java `interface` is similar to an abstract class; it cannot be instantiated to create objects.

- subtype polymorphism

Java's the system treats child classes as subtypes of the parent class. It dynamically binds the invocation of a method to the appropriate method implementation by searching up the inheritance hierarchy.

As we use Java in this chapter, a Java object exhibits all three "essential characteristics" of objects [5]:

a. state

   The state of a Java object is the mapping of the object's attributes (i.e., instance variables) to their values.

b. operations

   The operations of a Java object are the methods defined for the object. A method takes the object and zero or more other arguments and either returns information about the object's state or changes the its state.

c. identity

   Each Java object (i.e., instance of a class) has an identity that is distinct from all other Java objects (e.g., the address of the object in memory).

As we use Java in this chapter, a Java object also exhibits both "important but non-essential characteristics" of objects [5]:

d. encapsulation

   A Java class defines sets of instance variables and methods for instances of the class. If these instance variables or methods are declared as `private`, then they can only be accessed from within the class and, hence, are encapsulated within the class. If they are marked as `public`, then they can be accessed from outside the class. As we see, we normally explicitly declare all instance variables `private` and all operations `public`.

   Java's encapsulation is thus at the class-level not the object level.

e. independent lifecycle

   Java objects exist independently of the program units (e.g., methods or classes) that create them. An object can be instantiated in one program unit and passed to other program units where it is used. When the object is no longer accessible, its resources (e.g., memory) can be reclaimed by the garbage collector.

This chapter uses a cluster of related Java classes to implement modules. Java classes should thus be designed and implemented to satisfy the principle of information hiding. That is, they should not reveal details of their internal implementation (i.e., their secrets). They should only reveal aspects needed for the abstraction.

According to the concepts and terminology used in the Types [2] chapter of the ELIFP textbook [4, Ch. 5], Java exhibits

- static typing

  The Java compiler determines the type of an expression based on the explicitly declared (or inferred) types of its component subexpressions.

- nominal typing

  In addition to the builtin types, the `class` and `interface` define types.

  If class `Child extends Parent`, then type `Child` is considered a subtype of type `Parent`. Similarly, if a class or interface `Child implements IParent`, then type `Child` is considered a subtype of type `IParent`.

- polymorphic variables

  Java variables declared with some type `Base` can hold objects of type `Base` or of any of its subtypes.

- polymorphic operations

  Java dynamically binds method invocations to method implementations along the subtype (inheritance) hierarchy.

  Java also overloads function calls and some builtin operators.

Java classes (especially those implementing ADTs) should be designed and implemented so that they satisfy the Liskov Substitution Principle [2,11,19]. That is, it should be possible to substitute a subclass instance for a superclass instance in any circumstances.

In the following sections, we examine how to use Java to implement abstract data types. We will elaborate on some of the concepts mentioned in this section.

2.3 Java Classes

As a language construct, a Java `class`{.java] is similar to a user-defined `struct` type in C or user-defined record *type* in Pascal. A `class` is a template for constructing data items that have the same structure but differing values (states). We say that an item constructed by a class is a *class instance* (or an *object*).

Like the C structure type or Pascal record type, a Java class can consist of several components. In C and Pascal, all the components are data fields. However, in Java, functions and procedures may be included as components of a class. These procedures and functions are called *methods*.

2.3.1 Class and Instance Methods

A method declared in a class may be either a class method or instance method.

- A *class method* is associated with the class as a whole, not with any specific instance.
- An *instance method* is associated with an instance of the class.

We declare a method as a *class method* by giving the keyword `static` in the header of its definition. For example, a `main` method of a program is a class method of the class in which it is defined.

```
public static void main(String[] args)
{   //  beginning code for the program
}
```

If we *do not include* the keyword `static` in the header of a method definition, the method is an *instance method*. For example, consider methods that implement the bounded stack's `push` and `top` operations as specified in the chapter Data Abstraction Concepts [3]:

```
public void pop()
{   // code for pop operation
}

public Object top()
{   // code for top operation
}
```

Note that `pop()` is a *procedure* (i.e., it has return type `void`) method and `top` is a *function* method.

Scala note: The Scala language [14,17] does not have static members of classes. However, the methods of a Scala singleton `object` have basically the same characteristics described above for "class methods". Often the "class methods" appear in the companion `object` for a class. i.e., the `object` with the same name as the class.

2.3.2 Class and Instance Variables

In a similar fashion, the variables (data fields) declared in a class may be either class variables or instance variables.

- A *class variable* is associated with the class as a whole; there is only one copy of the variable for the entire class. As with methods, the keyword `static` is used to declare a class variable.
- An *instance variable* is associated with an instance of the class; each instance has its own instance of the variable. As with methods, the absence of the keyword `static` denotes an instance variable.

An instance method has direct access to the instance variables of the class instance (object) to which it is applied. The instance's variables are implicit arguments of the method calls. (If needed to distinguish among names, the builtin variable `this` can be used to refer to the instance to which the method is applied.) The instance methods also have access to the class variables (if any).

Class methods only have access to the class variables. The methods do not have any implicit arguments. In fact, class methods can be called without any instances of the class being in existence.

Scala note: The Scala language does not have static members of classes. However, the variables of a Scala singleton `object` have basically the same characteristics described above for "class variables". Often the "class variables" appear in the companion `object` for a class. i.e., the `object` with the same name as the class.

2.3.3 Public and Private Accessibility

The components of a class can be designated as `public` or `private`.

- The `public` components of the class are accessible from anywhere in the program (i.e., from any package).
- The `private` components are only accessible from inside the class.

As a general rule, the data fields of a class should be *private instance variables*, meaning that they are associated with a specific instance and are only accessible by the instance methods. This hides, or encapsulates, the data fields within the class instance.

Note: Actually, the instance methods of a Java class can access the instance variables of any instance of that class, not just the current instance.

In general, avoid public instance variables. They break the principle of information hiding, leading to potential entanglements among modules.

A public method of a class is a service provided by that instance to other parts of a program. The private methods of a class can be used in implementing the public methods.

Class methods and variables should be used sparingly. These are more or less the types of subprograms and global variables found in languages like C and Pascal. Their excessive use can greatly reduce the potential benefits that can be realized from object-oriented techniques.

Java note: There are two other types of accessibility, "friendly" and `protected`, but `public` and `private` are sufficient for our discussion of ADT implementations.

Scala note: Although similar in concept to that of Java, the accessibility features of Scala differ somewhat [14,17]. By default, all features are public in Scala, but accessibility can restricted in a more fine-grained manner than in Java. The unmodified keyword `private` has the same meaning as in Java.

2.3.4 Primitive and Reference Variables

A Java variable is a strongly typed "container" in memory that is declared to hold either:

- a *value* of the associated primitive data type such as integers (`int`), floating point numbers (`double`), booleans (`boolean`), and single characters (`char`).
- a *reference* to (i.e., memory address of) an instance of the associated class (or other reference) type.

Java note: Although arrays are not class instances, array variables hold a reference to an instance of the array.

The class instances themselves are stored in the dynamically managed heap memory area. Java allocates memory from the heap to hold newly constructed instances of a class. Java's garbage collector reclaims the memory for instances that are no longer needed by the program.

Note: Recent versions of Java can sometimes hide the differences between primitive values and references by automatically "boxing" primitive values as instances of the corresponding wrapper classes (e.g., `int` values as `Integer` instances). Scala goes further in that primitives and references are in the same type hierarchy. However, both languages run on the Java Virtual Machine, which makes a distinction between primitive values and references (i.e., pointers), so it is not possible to avoid the distinction entirely.

2.4 Implementing ADTs as Java Classes

If only one implementation of an ADT is needed, the following techniques can be used to implement an ADT using Java.

The implementation techniques discussed in this section implement the ADT in an imperative way. That is, instead of returning a new instance of the ADT with a modified state, a mutator operation usually modifies the state of the existing instance.

Caveat: The discussion of Java in this chapter does not use generic type parameters. For the `StackB` ADT (defined in the Data Abstraction Concepts), the type of the `Item` values stored in the stack can be a parameter of the `StackB` class.

TODO: Consider modifying this discusion to use an `Item` generic parameter.

1. **Use the Java `class` construct to represent the entire ADT.** If we want to allow access to the class from anywhere in the program, we will make the class `public`.

   For the `StackB` ADT, we can use the following structure for the class:

   ```
   public class StackB
   {   // implementation of instance methods and data here
   }
   ```

2. **Use an instance of the Java class to represent an instance of the ADT and, hence, variables of the class type to hold references to instances.**

For example, to declare a variable that can hold a reference to a `StackB` instance, we can use the following declaration:

```
StackB stk;
```

3. **As each component of the class is defined, ensure that the semantics of the ADT operations are implemented appropriately.** That is, make sure:

   - an appropriate implementation (representation) invariant is defined to capture what it means for the internal state of an instance to be valid,
   - the interface and implementation invariants are established (i.e., made true) by the constructors and preserved (i.e., kept true) by the mutator and accessor methods,
   - each method's postcondition is established by the method in any circumstance when it is called with the precondition true.

   The class and its methods should be documented with the invariants, preconditions, and postconditions.

4. **Represent the ADT's constructors by Java constructor methods. In most circumstances, also include a parameterless default constructor.**

   A Java constructor is a method with the same name as the class. It does not have a return type specified. Upon creation of an instance of the class, the constructor initializes the instance's state so that the class invariants are established.

   A constructor is normally invoked by the Java operator `new`. The operator `new` allocates memory on the heap for the instance, calls the constructor to initialize the new instance, and then returns a reference to the new instance.

   For example, we can represent the ADT operation `create` by the constructor method `StackB`.

```
public class StackB
{   public StackB(int size)
    {   // initialization code
    }

// rest of StackB methods and data ...
}
```

   A user of the `StackB` class can then declare a variable and initialize it to hold a reference to a new stack with a capacity of 100 items as follows:

```
StackB stk = new StackB(100);
```

The expression `new StackB(100)` allocates a `StackB` instance in the heap storage and calls the constructor above to initialize the data fields encapsulated within the instance.

5. **Represent the ADT operations by instance methods of the class.** Thus the state of the ADT instance, which is given explicitly in the ADT signatures, becomes an *implicit argument* of all method calls. Mutators also have the state as an implicit return.

   We can apply a method to a class instance by using the selector (i.e., "dot") notation. This notation is similar to the notation for accessing `record` components in Pascal.

   For example, in the case of the `StackB` ADT we can represent the operations as instance methods of class `StackB`. The explicit `StackB` parameters and return values of the operations thus become implicit.

   Suppose we want to push an item `x` onto the `stk` created above. We can do that with the following code:

   ```
   if (!stk.full())
       stk.push(x);
   ```

   We can then examine the top item and remove it:

   ```
   if (!stk.empty())
   {   it = stk.top();
       stk.pop();
   }
   ```

6. **Make the constructors, mutators, accessors, and destructors `public` methods of the class.** That is, precede the method's definition by the keyword `public`.

7. **Represent the ADT mutator operations by Java procedure (i.e., `void`) methods, except those mutator operations that explicitly require new instances to be generated (e.g., a copy or `clone` operation).**

   For example, the `pop` method of `StackB` would have the following structure:

   ```
   public void pop()
   {   //  code to implement operation
   }
   ```

   A mutator method modifies the encapsulated state of the class instance (which is the implicit argument of the method). In any circumstance in which its precondition and the class invariants hold on entry, the method must establish its postcondition and reestablish the invariants upon exit. (The invariant might not hold in the middle of the method's execution.)

Comment: Implementing mutator operations as procedure calls that modify the stored state is really an optimization. All mutators can be implemented in the applicative style, returning a modified copy of the instance. This implementation might, however, be inefficient in use of processor time and memory.

8. **For certain mutator operations (e.g., copy or `clone`), implement the corresponding Java methods to return new instances of the class rather than to modify the current instance (i.e., their implicit arguments).**

   Any mutator method must, of course, establish its postcondition and reestablish the invariants for the current instance. In addition, these applicative mutators must also establish the invariants for the new instance returned.

9. **Represent the ADT accessor operations by Java function methods of < the proper return type.**

   For example, the `empty` method of `StackB` would have the following structure:

   ```
   public boolean empty()
   {   //  code to implement operation
   }
   ```

   An accessor method accesses the encapsulated state of the class instance (which is the implicit argument) and computes a value to be returned. In any circumstance in which its precondition and the class invariants hold on entry, the method must establish its postcondition and reestablish the invariants upon exit. (The invariant might not hold in the middle of the method's execution.)

10. **If necessary for deallocation of internal resources, represent the ADT destructor methods by explicit Java procedures; in most cases, however. just allow the automatic garbage collection to reclaim instances that are no longer being used.**

    For example, in the `StackB` class, we might include an explicit destroy operation that releases the storage resources and disables further use of the instance.

    ```
    public void destroy()
    {   //  code to free resources
    }
    ```

    Java note: The Java framework allows a `finalize()` method to be included in each class. This method is called implicitly whenever the garbage collector detects that the instance is no longer in use. However, since it is difficult to predict when (if ever) this method will be executed, it is safer to include explicit destructors when resources are in short supply and must be explicitly managed.

11. **Use `private` data fields of the Java class to represent the encapsulated state of the instance needed for a particular implementation.** By making the data fields `private` they are still available to the instance's methods, but are not visible outside the class.

    For example, the `StackB` class might have the following data fields:

    ```
    public class StackB
    {   //  public operations of class instance

        //  encapsulated data fields of class instance
        private int topItem;    // Pointer to next index for insertion
        private int capacity;   // Maximum number of items in stack
        private Object[] stk;   // the stack
    }
    ```

12. **Do not use `public` data fields in the class. These violate the principle of information hiding. Instead introduce appropriate accessor and mutator methods to allow manipulation of the hidden state.**

13. **Include, as appropriate, `private` methods to aid in implementation.**

    Functionality common to several methods can be placed in separate functions and procedures as needed. However, since these are `private`, they can only be accessed from within the class and thus can be changed without affecting the public interface of the class.

14. **Add any other methods needed to make the ADT fit into the Java environment.**

    For example, it is frequently useful to add public `toString` and `clone` methods. The `toString` method returns a Java `String` reflecting the "value" of the instance in a format suitable for printing. The `clone` method creates a new instance that has the same value as the current instance.

15. **In general, avoid use of class (i.e., `static`) variables.** Since a class variable is shared among all instances of the class, it may be difficult to preserve the invariants for individual instances as the value of the class variable changes.

    However, it is a good programming practice to **use class *constants* where appropriate.** These are data fields declared with both the `static` and `final` modifiers. Their values may be initialized but cannot be changed thereafter.

These constants may be declared `private` if usage is to be restricted to the class or `public` if the users of the class also need access.

By convention, the names of constants are normally written with all uppercase letters. For example, the following defines a symbolic name for the integer code used for Sunday as a day of the week in the `Day` class defined later.

```
public static final int SUNDAY = 1;
```

Caveat: When this set of notes was originally written, Java did not yet have generics. So the examples below handle the type parameters of the ADT in other ways. A Java generic provides a class facility that can be parameterized with types (like the C++ `template` or Ada `generic` mechanisms).

For example, in the implementation below we represent the set `Item` of the `StackB`{.java} ADT by the class `Object`. As we will see when we discuss inheritance, the `Object` type will allow us to store an instance of any class on the `StackB`. With this definition, any data of a reference type can appear in the stack, but values of the primitive types cannot. A better implementation would have `Item` as type parameter of the class.

The next section gives a Java implementation of the `StackB` ADT. A similar constructive definition and two implementations of a Queue ADT are available in a separate document.

TODO: Beter integrate SoftwareInterfaces into this document collection.

2.4.1 Java Implementation of Bounded Stack

TODO: Reconstruct (or find) the following source code and ensure that it executes on the current Java platform. Insert link.

In this section, we give an implementation of the `StackB` ADT that uses an array of objects and an integer "pointer" to represent the stack. (This implementation does not use Java generic classes.)

This implementation is not robust; each operation assumes that its precondition holds. A more robust implementation might check whether the precondition holds and throw an exception if it does not.

Remember that the invariants are implicitly pre- and postconditions of all mutator and accessor methods, postconditions of the constructor, and preconditions of the destructor.

```
// A Bounded Stack ADT
public class StackB
{   // Interface Invariant:  Once created and until destroyed, this
    //      stack instance has a valid and consistent internal state

    public StackB(int size)
```

```
// Pre:   size >= 0
// Post:  initialized new instance with capacity size && empty()
{   stk = new Object[size];
    capacity = size;
topItem = 0;
}

public void push(Object item)
// Pre:   NOT full()
// Post:  item added as the new top of this instance's stack
{   stk[topItem] = item;
    topItem++;
}

public void pop()
// Pre:   NOT empty()
// Post:  item at top of stack removed from this instance
{   topItem--;
    stk[topItem] = null;
}

public Object top()
// Pre:   NOT empty()
// Post:  return item at top of this instance's stack
{   return stk[topItem-1];
}

public boolean empty()
// Pre:   true
// Post:  return true iff this instance's stack has no elements
{   return (topItem <= 0);
}

public boolean full()
// Pre:   true
// Post:  return true iff this instance's stack is at full capacity
{  return (topItem >= capacity);
}

public void destroy()
// Pre:   true
// Post:  internal resources released;  stack effectively deleted
{   stk = null;
capacity = 0;
    topItem = 0;
}
```

```
    // Implementation Invariant for informal model:
    //     0 <= topItem <= capacity &&
    //     stack is in array section stk[0..topItem-1]
    //         with the top at stk[topItem-1], etc.

    // Implementation Invariant for more formal model representing stack
    // as tuple (integer max, sequence stkseq)
    //     m == capacity && 0 <= topItem <= capacity &&
    //     stackInArray(stk,topItem,stkseq)
    //     where stackInArray(arr,t,ss) = if t == 0 then ss == []
    //                                    else arr[t-1] == head(ss)
    //                                       && stackInArray(arr,t-1,tail(ss))

    private int topItem;   // Pointer to next index for insertion
    private int capacity;  // Maximum number of items in stack
    private Object[] stk;  // the stack
}
```

2.5 Better Approach to Implementing ADTs in Java

TODO: Reconstruct (or find) the following source code and ensure that it executes on the current Java platform. Inset link.

If several different implementations of an ADT are needed, then the Java specification of an ADT's interface should be separated from the class implementation. The interface specification can be reused among several classes and various implementations of the interface can be used interchangeably.

This can be done as follows.

1. **Define a Java `interface` that specifies the type signatures for the ADT's mutator and accessor (and, if needed, destructor) operations.** These method signatures should have the same characteristics as described above in the discussion of class-based specification.

2. **Specify and document the `interface` by the interface invariants, preconditions, and postconditions that must be supported by any implementation of the ADT.** There are no implementation invariants for an interface, but individual classes that implement the `interface` will have them.

   For example, a bounded stack `interface` might be specified as follows:

   ```
   public interface StackADT
   {   // Interface Invariant:  Once created and until destroyed, this
       //     stack instance has a valid and consistent internal state

       public void push(Object item);
   ```

```
        // Pre:   NOT full()
        // Post:  item added as the new top of this instance's stack

        ...

        public Object top();
        // Pre:   NOT empty()
        // Post:  return item at top of this instance's stack

        ...
    }
```

3. **Provide one or more concrete classes that implement the `interface`.**

   For example, an array-based `StackADT` could be implemented similarly to the `StackB` definition given in the previous section.

```
    public class StackInArray implements StackADT
    {   // Interface Invariant:  Once created and until destroyed, this
        //      stack instance has a valid and consistent internal state

    public StackInArray(int size)
    // Pre:   size >= 0
    // Post:  initialized new instance with capacity size && empty()
    {   stk = new Object[size];
    capacity = size;
             topItem = 0;
        }

        public void push(Object item)
        // Pre:   NOT full()
        // Post:  item added as the new top of this instance's stack
        {   stk[topItem] = item;
            topItem++;
        }

        ...

        public Object top()
        // Pre:   NOT empty()
        // Post:  return item at top of this instance's stack
        {   return stk[topItem-1];
        }

        ...
    // Implementation Invariant for informal model:
    //     0 <= topItem <= capacity &&
    //     stack is in array section stk[0..topItem-1]
    //         with the top at stk[topItem-1], etc.

    // Implementation Invariant for more formal model representing stack
    // as tuple (integer max, sequence stkseq)
    //     m == capacity && 0 <= topItem <= capacity &&
    //     stackInArray(stk,topItem,stkseq)
    //     where stackInArray(arr,t,ss) =
    //               if t == 0 then ss == []
    //               else arr[t-1] == head(ss)
    //                    && stackInArray(arr,t-1,tail(ss))

    private int topItem;   // Pointer to next index for insertion
    private int capacity;  // Maximum number of items in stack
    private Object[] stk;  // the stack
}
```

4. **Declare variables of the ADT's `interface` type to hold instances of any concrete class that implements the `interface`.** Any of the operations defined in the `interface` can be applied to the instance to which this variable refers.

   For example, a variable of type `StackADT` can hold instances of any concrete class that implements the `interface StackADT`.

   ```
   StackADT theStack = new StackInArray(100);
   theStack.push("Hello World");
   ```

For an ADT specification and implementations that follow this approach, see the description of the Ranked Sequence ADT case study given in a separate document. In addition to Java interfaces, the Ranked Sequence case study uses other Java features such as exceptions, enumerations, packages, and Javadoc annotations.

TODO: Integrate SoftwareInterfaces into this document collection.

2.5.1 Java Class Implementation for Day

TODO: Reconstruct (or find) the following source code and ensure that it executes on the current Java platform. Inset link.

The following implementation of the `Day` ADT is adapted from the like-named class in Horstmann and Cornell's book *Core Java* [10].

This implementation represents the calendar as three integers. It converts the dates to and from Julian dates to do some of the operations.

```
//  This class implementation is adapted from the Day class in
//  Horstmann and Cornell, Core Java 1.2: Volume I - Fundamentals
//  (Fourth Edition), Prentice Hall, 1999.

import java.util.*;
import java.io.*;

public class Day
{
    // Interface Invariant:  Once created and until destroyed, this
    //    instance contains a valid date.  getdate() != 0 &&
    //    1 <= getMonth() <= 12 && 1 <= getDay() <= #days in getMonth().
    //    Also calendar date getMonth()/getDay()/getYear() does not
    //    fall in the gap formed by the change to the modern
    //    (Gregorian) calendar.

  //  Constants for days of the week

    public static final int SUNDAY    = 1;
    public static final int MONDAY    = 2;
    public static final int TUESDAY   = 3;
    public static final int WEDNESDAY = 4;
    public static final int THURSDAY  = 5;
    public static final int FRIDAY    = 6;
    public static final int SATURDAY  = 7;

  // Constructors

    public Day()
    //  Pre:   true
    //  Post:  the new instance's day, month, and year set to today's
    //             date (i.e., the date of creation of the instance)
    //
    //  Implementation uses GregorianCalendar class from the Java API
    //  to get today's date.
    //
    {   GregorianCalendar todaysDate = new GregorianCalendar();
        year  = todaysDate.get(Calendar.YEAR);
        month = todaysDate.get(Calendar.MONTH) + 1;
        day   = todaysDate.get(Calendar.DAY_OF_MONTH);
    }

    public Day(int y, int m, int d)
           throws IllegalArgumentException
    //  Pre:   y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month m
    //         (y,m,d) does not fall in the gap formed by the
  //             change to the modern (Gregorian) calendar.
  //  Post:  the new instance's day, month, and year set to y, m,
  //         and d, respectively
  //  Exception:  IllegalArgumentException if y m d not a valid date
  {   year  = y;
      month = m;
      day   = d;
      if (!isValid())
          throw new IllegalArgumentException();
  }

// Mutators

  public void setDay(int y, int m, int d)
              throws IllegalArgumentException
  //  Pre:   y != 0 && 1 <= m <= 12 && 1 <= d <= #days in month m
  //         (y,m,d) does not fall in the gap formed by the
  //             change to the modern (Gregorian) calendar.
  //  Post:  this instance's day, month, and year set to y, m,
  //         and d, respectively
  //  Exception:  IllegalArgumentException if y m d not a valid date
  {   year  = y;
      month = m;
      day   = d;
      if (!isValid())
          throw new IllegalArgumentException();
  }

  public void advance(int n)
  //  Pre:   true
  //  Post:  this instance's date moved n days later.  (Negative n
  //             moves to an earlier date.)
  {   fromJulian(toJulian() + n);
  }

// Accessors

  public int getDay()
  //  Pre:   true
  //  Post:  returns the day from this instance, where
  //             1 <= getDay() <= #days in this instance's month
  {   return day;
  }

  public int getMonth()
  //  Pre:   true
//  Post:  returns the month from this instance's date, where
//              1 <= getMonth() <= 12
{   return month;
}

public int getYear()
//  Pre:   true
//  Post:  returns the year from this instance's date, where
//              getYear() != 0
{   return year;
}

public int getWeekday()
// Pre:   true
// Post:  returns the day of the week upon which this instance
//             falls, where 1 <= getWeekday() <= 7;
//             1 == Sunday, 2 == Monday, ..., 7 == Saturday
{   //  calculate day of week
    return (toJulian() + 1) % 7 + 1;
}

public boolean equals(Day dd)
//  Pre:   dd is a valid instance of Day
//  Post:  returns true if and only if this instance and instance
//              dd denote the same calendar date
{   return (year == dd.getYear() && month == dd.getMonth()
            && day == dd.getDay());
}

public int daysBetween(Day dd)
//  Pre:   dd is a valid instance of Day
//  Post:  returns the number of calendar days from the dd
//              instance's date to this instance's date, where
//              equals(dd.advance(n)) would hold
{   //  implementation code
    return toJulian() - dd.toJulian();
}

public String toString()
//  Pre:   true
//  Post:  returns this instance's date expressed in the format
//              "Day[year,month,day]"
{
    return "Day[" + year + "," + month + "," + day + "]";
}
```

```
//  Destructors -- None needed

//  Private Methods

  private boolean isValid()
  //  Pre:   true
  //  Post:  returns true iff this is a valid date
  {   Day t = new Day();
      t.fromJulian(this.toJulian());
      return t.day == day && t.month == month
             && t.year == year;
  }

  private int toJulian()
  //  Pre:   true
  //  Post:  returns Julian day number that begins at noon of this day
  //
  //  A positive year signifies A.D., negative year B.C.
  //  Remember that the year after 1 B.C. was 1 A.D. (i.e., no year 0).
  //
  //  A convenient reference point is that May 23, 1968, at noon
  //  is Julian day 2440000.
  //
  //  Julian day 0 is a Monday.
  //
  //  This algorithm is from Press et al., Numerical Recipes
  //  in C, 2nd ed., Cambridge University Press 1992.
  //
  {   int jy = year;
      if (year < 0)
          jy++;
      int jm = month;
      if (month > 2)
          jm++;
      else
      {   jy--;
          jm += 13;
      }
      int jul = (int) (java.lang.Math.floor(365.25 * jy)
                + java.lang.Math.floor(30.6001*jm) + day + 1720995.0);

      int IGREG = 15 + 31*(10+12*1582);
          // Gregorian Calendar adopted Oct. 15, 1582

      if (day + 31 * (month + 12 * year) >= IGREG)
         // change over to Gregorian calendar
```

```
    {   int ja = (int)(0.01 * jy);
        jul += 2 - ja + (int)(0.25 * ja);
    }
    return jul;
}

private void fromJulian(int j)
//  Pre:   true
//  Post:  this calendar Day is set to Julian date j
//
//  This algorithm is from Press et al., Numerical Recipes
//  in C, 2nd ed., Cambridge University Press 1992
//
{   int ja = j;

    int JGREG = 2299161;
        /* the Julian date of the adoption of the Gregorian
           calendar
        */

    if (j >= JGREG)
    /* correct for crossover to Gregorian Calendar */
    {   int jalpha = (int)(((float)(j - 1867216) - 0.25)
            / 36524.25);
        ja += 1 + jalpha - (int)(0.25 * jalpha);
    }
    int jb = ja + 1524;
    int jc = (int)(6680.0 + ((float)(jb-2439870) - 122.1)
             /365.25);
    int jd = (int)(365 * jc + (0.25 * jc));
    int je = (int)((jb - jd)/30.6001);
    day = jb - jd - (int)(30.6001 * je);
    month = je - 1;
    if (month > 12)
        month -= 12;
    year = jc - 4715;
    if (month > 2)
        --year;
    if (year <= 0)
        --year;
}

//  Implementation Invariants:
//      year != 0 && 1 <= month <= 12 && 1 <= day <= #days in month
//      (year,month,day) not in gap formed by the change to the
//      modern (Gregorian) calendar
```

```
        private int year;
        private int month;
        private int day;
    }
```

2.6 What Next?

TODO

2.7 Exercises

TODO

2.8 Acknowledgments

This chapter was originally part of my Data Abstraction Concepts notes [3]. See the Acknowledgments section of that chapter for more information on its development. For the Lua-based offering of CSci 658 in Fall 2013. I separated most of the Java-specific material into this chapter to make the content of the Data Abstraction chapter more language independent.

In this chapter, I use a Java programming style influenced by Horstmann and Cornell's book *Core Java* [10]. I adapted the Java `Day` design and implementation from the like-named class in Chapter 4 of Horstmann and Cornell's of that book [10].

In Summer 2017, I modified this chapter slightly to link it into the revised (Pandoc Markdown) version of the Notes on Data Abstraction chapter and then reformatted it to use Pandoc Markdown in Spring 2018.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

In 2022, I also added the section "Java as an Object-Oriented Language" to better tie this chapter to the concepts and terminology used in the ELIFP textbook [4], especially chapters 3 and 5.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

2.9 Concepts

TODO

2.10 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] H. Conrad Cunningham. 2019. *Type system concepts*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/TypeConcepts/TypeSystemConcepts.html

[3] H. Conrad Cunningham. 2022. *Data abstraction concepts*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DA01/DataAbstractionConcepts.html

[4] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[5] H. Conrad Cunningham. 2022. Object-based paradigms. In *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/Ch03/03_Object_Paradigms.html

[6] H. Conrad Cunningham. 2022. *Object-oriented software development*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/OOSoftDev/OOSoftDev.html

[7] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications*. D. C. Heath, Lexington, Massachusetts, USA.

[8] David Gries. 1981. *Science of programming*. Springer, New York, New York, USA.

[9] Cay S. Horstmann. 1995. *Mastering object-oriented design in C++*. Wiley, Indianapolis, Indiana, USA.

[10] Cay S. Horstmann and Gary Cornell. 1999. *Core Java 1.2: Volume I—Fundamentals*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[11] Barbara Liskov. 1987. Keynote address—Data abstraction and hierarchy. In *Proceedings on object-oriented programming systems, languages, and applications (OOPSLA '87): addendum*, ACM, Orlando, Florida, USA, 17–34.

[12] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[13] Hanspeter Mossenbock. 1995. *Object-oriented programming in Oberon-2*. Springer, Berlin, Germany.

[14] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[15] Martin Odersky, Lex Spoon, and Bill Venners. 2021. *Programming in Scala* (Fifth ed.). Artima, Inc., Walnut Creek, California, USA.

[16] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15, 12 (December 1972), 1053–1058.

[17] Scala Language Organization. 2022. The Scala programming language. Retrieved from https://www.scala-lang.org/

[18] Pete Thomas and Ray Weedom. 1995. *Object-oriented programming in Eiffel*. Addison-Wesley, Boston, Massachusetts, USA.

[19] Wikpedia: The Free Encyclopedia. 2022. Liskov substitution principle. Retrieved from https://en.wikipedia.org/wiki/Liskov_substitution_principle