# CSci 658-01: Software Language Engineering Data Abstraction

**H. Conrad Cunningham**

**17 February 2018**

## Contents

Copyright (C) 2017, 2018, H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

**Advisory**: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of February 2018 is a recent version of Firefox from Mozilla.

TODO:

- Update the programming language support to be more generic.
- Add Exercises and Concepts sections

# Data Abstraction

## 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"*.

## Kinds of abstraction

Large complex systems can only be made understandable by decomposing them into modules. When viewed from the outside, from the standpoints of users, each *module* should be simple, with the complexity hidden inside.

We strive for modules that have simple interfaces that can be used without knowing the implementations. Here we use *interface* to mean any information about the module that other modules must assume to be able to do their work correctly.

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.

When we develop an algorithm following the top-down approach, we are practicing procedural abstraction. At a high level, we break the problem up into several tasks. We give each task a name and state its requirements, but we do not worry about how the task is to be accomplished until we expand it at a lower level of our design.

When we code a task in a programming language, we will typically make each task a subprogram (procedure, function, subroutine, method, etc.). Any other program component that calls the subprogram needs to know its interface (name, parameters, return value, assumptions, etc.) but does not need to know the subprogram's internal implementation details. The internal implementation can be changed without affecting the caller.

In data abstraction, the focus is on the problem's data rather than the tasks to be carried out.

## 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 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 object; the object may be in 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. Haskell also groups functions and other declarations into a program unit called a `module`. A `module` explicitly exports selected functions and keep others hidden.

## 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.

## Abstract Data Structures

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.

## 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.

## 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.

## 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.

### 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`.

### 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 }`

### 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.

#### Constructors

```
create:  -> Stack
```

#### Mutators

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

pop: Stack -> Stack
```

#### Accessors

```
top: Stack -> Item

empty: Stack -> boolean
```

#### 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.

**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.

## 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.

### Name

`StackB` (of `Item`)

### 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, ... }`

### Signatures

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

#### Constructors

```
create:  integer -> StackB
```

#### 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.

#### Accessors

```
top: StackB -> Item
```

```
empty: StackB -> boolean
```

```
full: StackB -> boolean
```

#### Destructors

```
destroy: StackB ->
```

### 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.

### 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')`

### 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

### 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.)

**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.

### 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.

#### 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.

#### 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.

**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`

**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 supplementary notes. A Java implementation of the StackB ADT appears in those notes.

## Date (Day) ADT

Consider an ADT for storing and manipulating calendar dates. We will 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 book *Core Java 1.2: Volume*

*I — Fundamentals* (Fourth Edition) by Cay S. Horstmann and Gary Cornell (Sun Microsystems Press/Prentice Hall, 1999).

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.

### 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`

### 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.)

### 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.

**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.

## 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)
```

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".

## 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.

## Exercises

TODO

## Acknowledgements

In Spring 2017 I adapted these lecture notes from my previous notes on this topic. The material here is based, in part, on the presentations in the following books:

- Nell Dale and Henry M. Walker. *Abstract Data Types: Specifications, Implementations, and Applications*, D. C. Heath,

1996.

- Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995.
- Cay S. Horstmann and Gary Cornell. *Core Java 1.2: Volume I – Fundamentals* (Fourth Edition) Sun Microsystems Press (Prentice-Hall), 1999.
- Bertrand Meyer. *Object-Oriented Program Construction*, Second Edition, Prentice Hall, 1997.
- Hanspeter Mossenbock. *Object-Oriented Programming in Oberon-2*, Springer-Verlag, 1995.
- Pete Thomas and Ray Weedom. *Object-Oriented Programming in Eiffel*, Addison-Wesley, 1995.

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 classes using 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. In Fall 2017 and Spring 2018, I revised the structure and text in minor ways.

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

## Concepts

TODO