
# CMPUT 175 Introduction to Foundations of Computing

ADT: Abstract Data Types

# Objectives

- In this lecture we will discuss Abstract Data Types.
- We will talk about data structures, the implementation of Abstract Data Types.
- We will learn how to create data types and describe their properties and operations independently of any implementation

# Data Types

- You are by now acquainted with data types like Integers, Strings, Booleans, and so on.

- To access the data, you use operations defined in the programming language (Python) for the data type.

- For instance by accessing list elements in Python, you would use the square bracket notation *myList[i]*

# Type Definitions

- Selecting a type for a variable determines the possible values for that variable; the operations that can be done on it; and the meaning of the data.
- We might want to define data with complex data structures that are not readily provided by the programming language
- We might want to enrich the behaviour (operations) of an existing data type.

# The need for more data types

- Software *evolve* as a result of new requirements or constraints.
- A modification to a program commonly requires a change in one or more of its data structures
- a new field might be added to a personnel record to keep track of more information; a list might be replaced by a some other structure to improve the program's efficiency

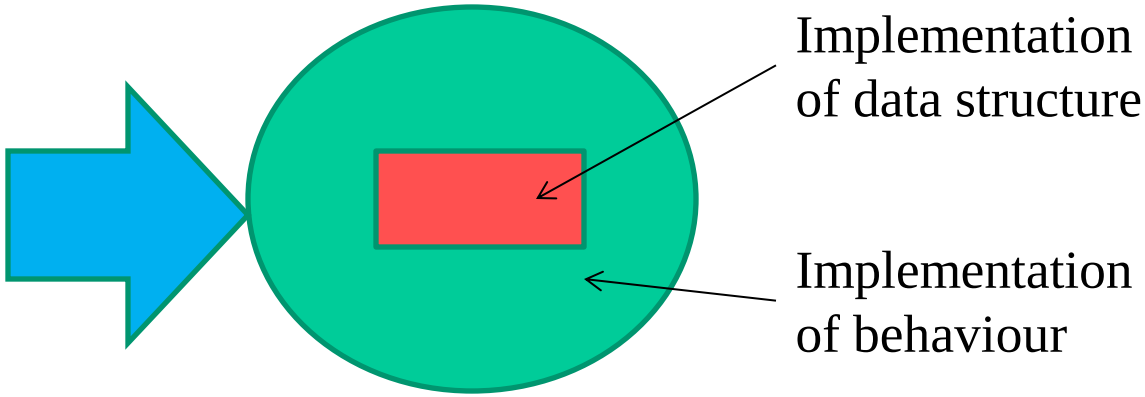
# The need for isolation

- If a data structure is changed or it's behaviour (interface) enhanced due to Software evolution or new requirements, maintenance of the software can become expensive if we need to revisit all the program
- By separating the implementation of a data structure and its behaviour from the implementation of the program that uses this data structure we minimize impact of change.
- You don't want a change in a data structure to require rewriting every procedure that uses the changed structure.

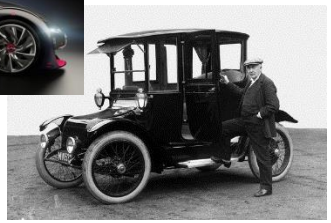
# Abstract data types

- What does ‘abstract’ mean?
- We make abstraction of the implementation of the data structure
- We hide the details of the implementation and show only the possible operations (interface)
- We talk about encapsulating the data structure

Access the data **via** interface (behaviour) and never directly to data structure implementation



# Concrete examples

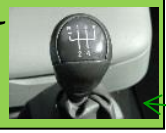


You don't need to know how the engine works, the mechanics of the gearbox, etc. in order to drive the car.

You don't need to know how calculators handle operations in order to use the calculator for even complex reckoning.

You don't need to know how the phone does establish connections and transmits voice in order to use it.

You need to know how to use the clutch and how to shift gears, etc.



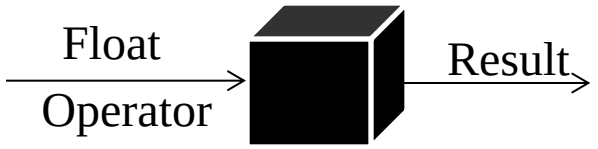
You need to know how to operate the provided buttons and functions, etc.

You need to know how to dial and where is the speakerphone and the microphone, etc.

**Interface**

# ADT Example: Floating point numbers

- You don't need to know how floating point arithmetic is implemented to use `float`
  - Indeed, the details can vary depending on processor, even virtual coprocessor
  - But the compiler hides all the details from you--some numeric ADTs are built-in
  - All you need to know is the syntax and meaning of operators, +, -, *, /, etc.



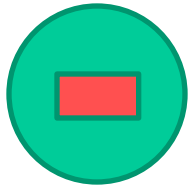

- Hiding the details of implementation is called **encapsulation** (**data hiding**)

# Other Existing encapsulated structures

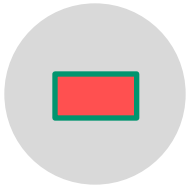
- Python provides a built in structure `list`
  - You do not need to know how Lists are implemented in order to use them.
  - All you need to know is the interface, the set of provided methods to use and manipulate the list

list.append(x) Add an item to the end of the list;
list.extend(L) Extend the list by appending all the items in the given list;
list.insert(i, x) Insert an item at a given position.
list.remove(x) Remove the first item from the list whose value is x.
list.pop([i]) Remove the last item in the list or the item at the given position in the list, and return it.
list.index(x) Return the index in the list of the first item whose value is x.
list.count(x) Return the number of times x appears in the list.
list.sort() Sort the items of the list, in place.
list.reverse() Reverse the elements of the list, in place.

# ADT = properties + operations

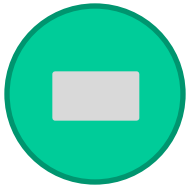


- An **ADT** describes a set of objects sharing the same properties and behaviors
  - The **properties** of an ADT are its **data** (representing the internal state of each object
    - `double d;` -- bits representing exponent & mantissa are its data or state



$12.987654321 \rightarrow 12987654321 \times 10^{-9}$

  - The **behaviors** of an ADT are its **operations** or **functions** (operations on each instance)
    - `sqrt(d) / 2;` //operators & functions are its behaviors



- Thus, an ADT couples its data and operations
  - OOP emphasizes **data abstraction**

# Formal, language-independent ADTs

- An ADT is a formal description, not code; independent of any programming language
  - *Why is code independence a good idea?*

- Promotes **design by contract**:
  - Specify responsibilities of suppliers and clients explicitly, so they can be enforced,  if necessary

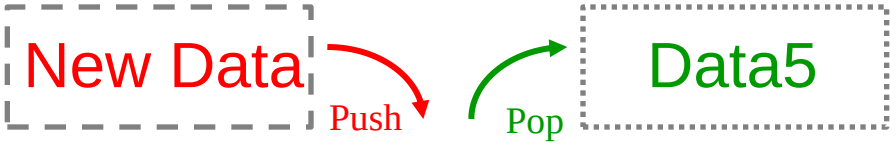
# Example: Bags and Sets

- Simplest ADT is a Bag (sets with duplication)
  - items can be added, removed, accessed
  - no implied order to the items
  - duplicates allowed

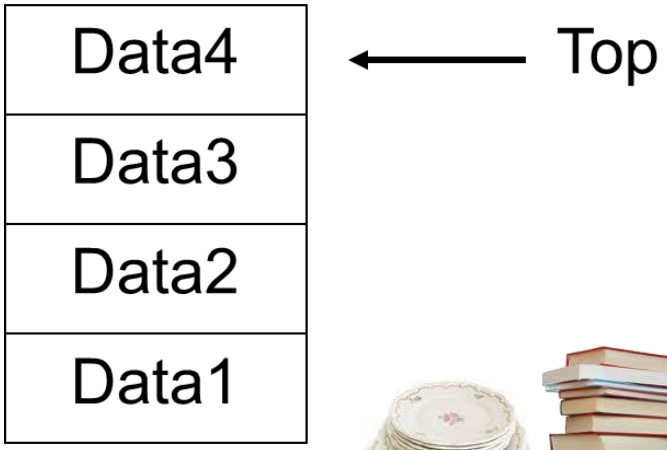
- Set
  - same as a bag, except duplicate elements are not allowed
  - union, intersection, difference, subset

# Example: Stacks

- Collection with access only to the last element inserted




- Last in - first out (LIFO)
  - Add element: push
  - Remove element: pop
  - top
  - is empty
  - make empty





Think of when you are browsing the web. The browser pushes the visited URLs on a stack. The back button pops them back.

# Other examples of ADT

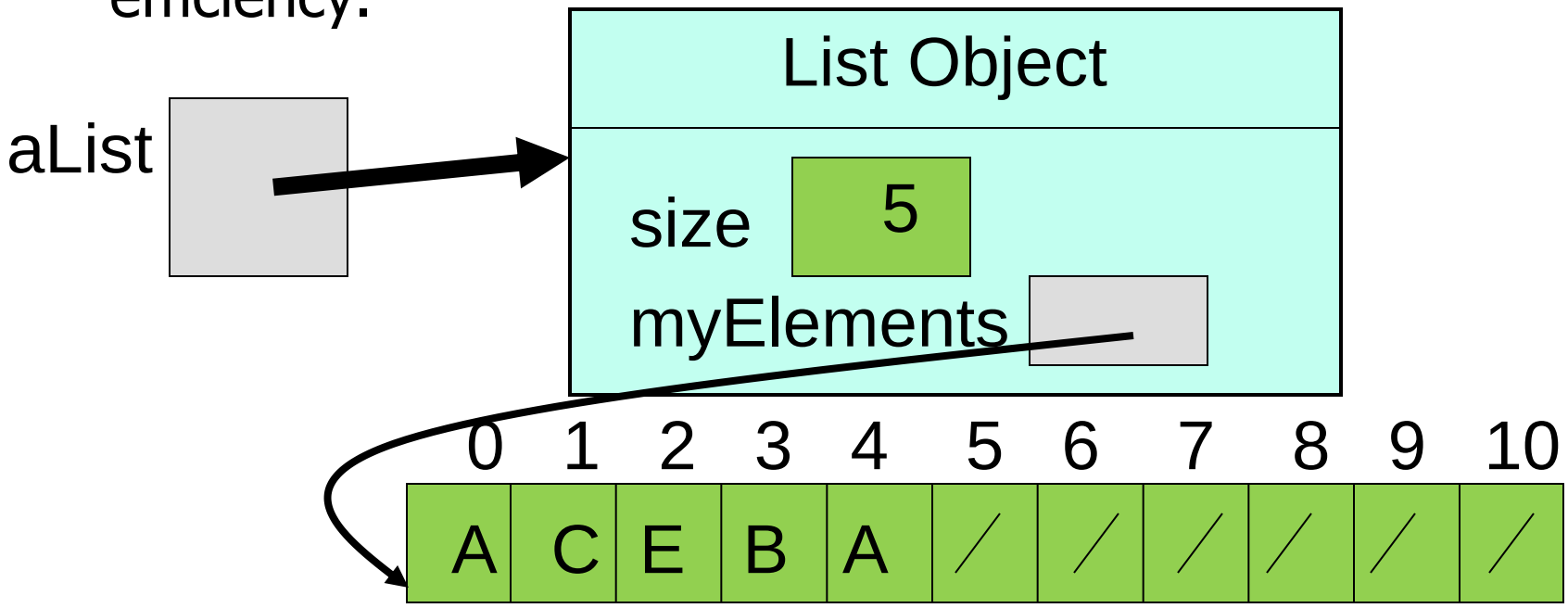
- Lists
- Queues
- Deque
- Pile (sorted deque)
- Trees
- Heaps
- Binary search trees
- B trees
- Maps
- Graphs
- …

# Abstraction and Implementation

- ADT represents a specification
  - Cannot be used directly in programming
- The physical implementation of an ADT is referred to at Data structure
  - Data Structure=Implementation of ADT
- In O-O programming, the implementation of choice for an ADT is the creation of a new class
  - Methods implement behaviour (operations)

# Data Structures

- A *Data Structure* is:
  - an implementation of an Abstract Data Type *and*
  - "An organization of information, usually in computer memory", for better algorithm efficiency."



# Data Structure Concepts

- Data Structures are containers:
  - they hold data
- Different types of data structures are optimized for certain types of operations
- Linear data structures are containers of data collections with ordered items. They differ on how items are added and removed.
- Linear structures have two ends: Front and Rear (or Top and Bottom) (or Head and Tail)
- Linear structures are containers that differ on the location their items are added or removed

# Core Operations

- Data Structures will have 3 core operations
  - a way to **add** things
  - a way to **remove** things
  - a way to **access** things

Mutators or setters

Accessors or getters

- Details of these operations depend on the data structure
  - Example: List, add at the end, access by location, remove by location
- More operations added depending on what data structure is designed to do

# Mutator and Accessor

- A **mutator method** is a method used to control changes to a variable
- It is also called a setter because it sets the value of a variable.
- Based on the principle of encapsulation a member variable of a class is made private to hide and protect it from other code.
- A public mutator method gets a new value as parameter, validates it (optionally) and assigns it to the private member variable.
- An **accessor** method returns the value of a member variable without altering it.
- It is also called a getter because it gets the value

# ADTs and Data Structures in Programming Languages

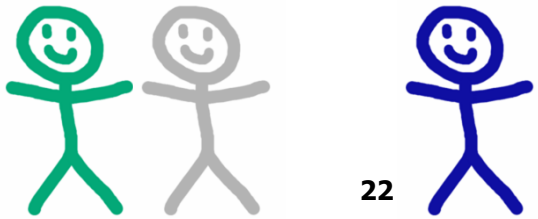
- Modern programming languages usually have a library of data structures
  - Java collections framework
  - C++ standard template library
  - Python lists, tuples, sets and dictionaries
  - Lisp lists

# Object Concepts

- An object has:
  - a private state
  - A set of resources that it provides. Let's call this its *public protocol*
- Each object is an instance of a class.
- The class defines the public protocol, so all objects of a class have the same protocol.
- The state of an object differentiates it from other objects in the same class.

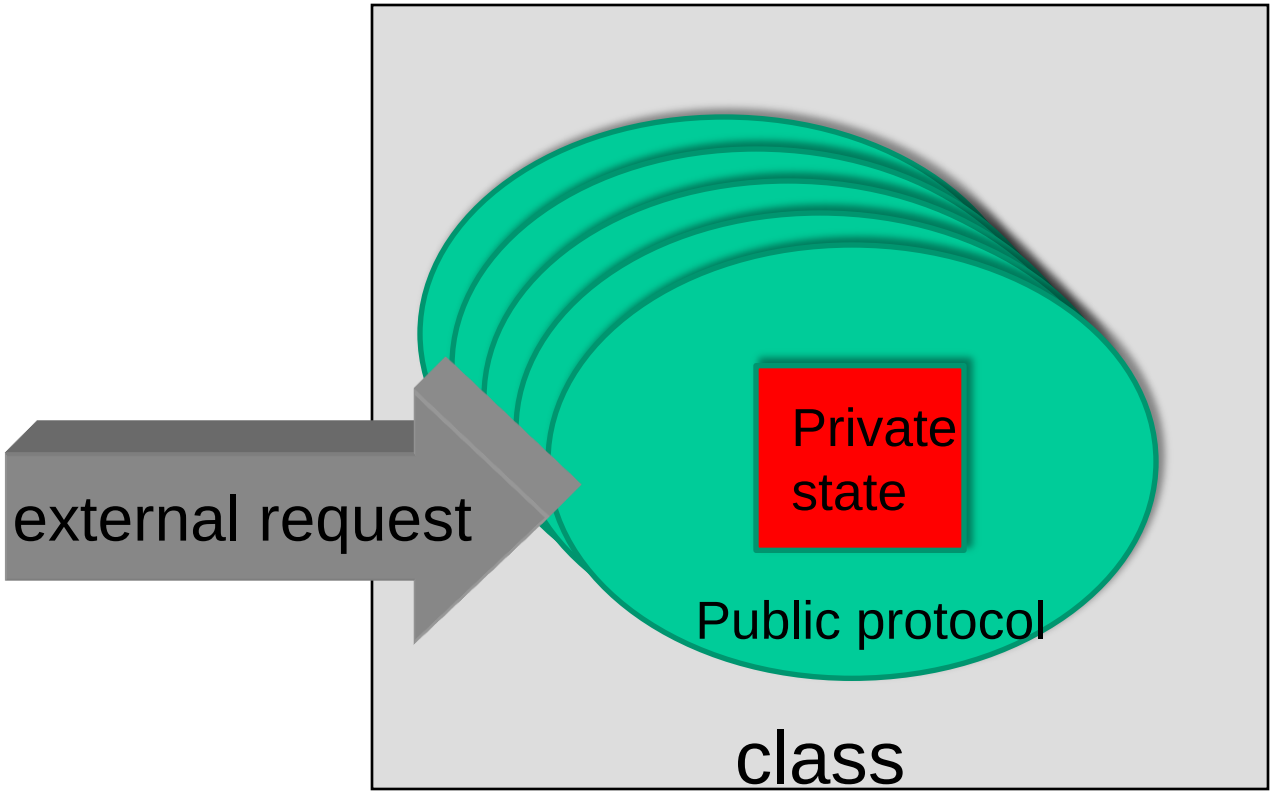
Remember:
The implementation of choice for an ADT is the creation of a new class.

Class stickman



Instances of class stickman

# Object Class, Protocol and State



# What getters and setters should not do

- To enforce encapsulation and data hiding, it is bad practice to have getter and setter methods expose the members properties (private state of objects).

- The exposed methods (public protocol) should act on the object to change its private state (setter or mutator) or act on the object to provide its data (getter or accessor) without exposing its implementation or properties.

- Outside the class, only the interface is known, while the implementation of the data structure is unknown (hidden).