Abstract Data Types

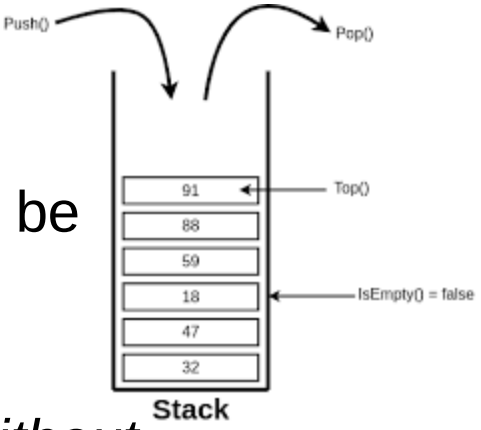
Object Oriented Programming (PCC-CS503) @UD

July, 2023

Content:

- Abstract data types and their specification.
- How to implement an ADT
- Concrete state space, concrete invariant, abstraction function.
- Implementing operations, illustrated by the Text example

Abstract Data Types (ADT)



- Represent data and operations performed on the data
- Focus: Data and allied operations but not how those will be implemented (~~memory organization, algorithms~~)
- Abstract: Provides implementation-independent view *(without referring a specific programming language)*
- ADT Examples:
  - List, Queue, Stack, e.g. stack can be implemented using built-in array declaration with push(), pop(), top() and isEmpty() oprations
  - Integers: For handling large integers (> 32 bits), structure implementation recommended in C *(use BigInteger class in Java):*

```
#define SIZEIDX 313
#define SIGNIDX 314
typedef unsigned long bigint[315];  // Handle 10000 bits. Bigger?
```

  - Similarly Real numbers, complex numbers, polynomials, matrices
  - Points: 2D and 3D points etc.

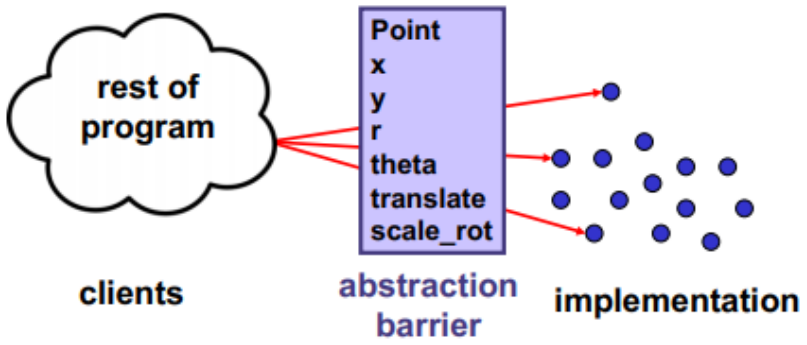
Specification / implemenation of Point ADT

```
class Point {
  // A 2-d point exists somewhere in the plane, ...
  public float x();
  public float y();
  public float r();
  public float theta();

  // ... can be created, ...
  public Point();          // new point at (0,0)
  public Point centroid(Set<Point> points);

  // ... can be moved, ...
  public void translate(float delta_x,
                        float delta_y);
  public void scaleAndRotate(float delta_r,
                             float delta_theta);
}
```

Observers

Creators/ Producers

Mutators



Specification of Point ADT:
Observers:
X: ...
Y: ...
R: ...
Theta: ...

Creators/Producers:

Specifying an ADT

Immutable

1. overview
2. abstract state
3. creators
4. observers
5. producers
6. ~~mutators~~

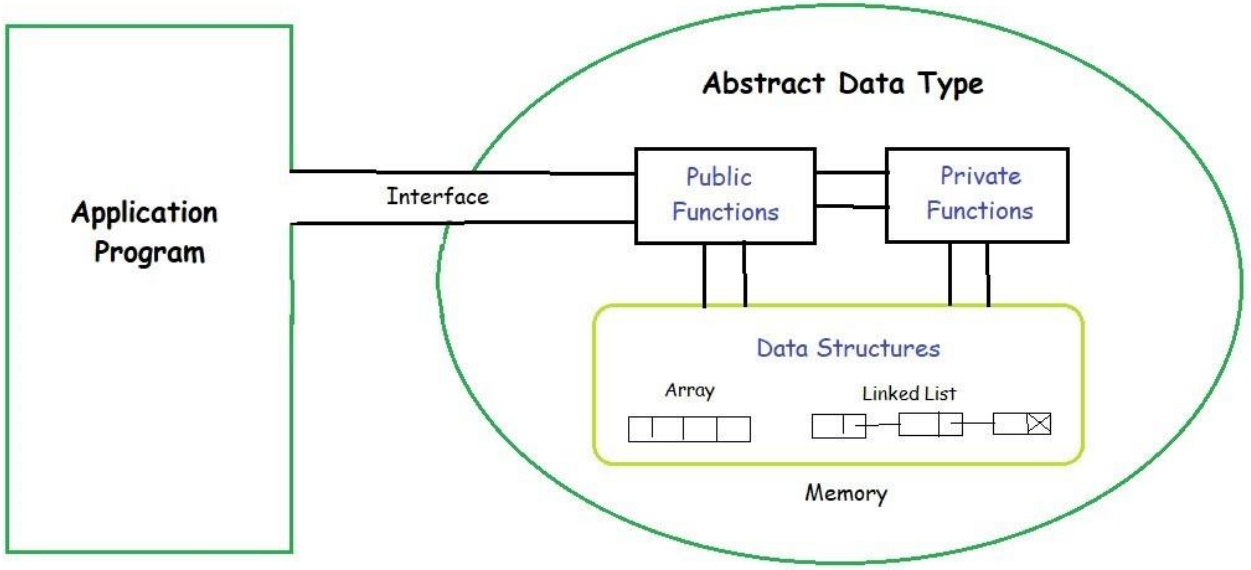
Mutable

1. overview
2. abstract state
3. creators
4. observers
5. producers (rare)
6. mutators

- Creators: return new ADT values (e.g., Java constructors)
- Producers: ADT operations that return new values
- Mutators: Modify a value of an ADT
- Observers: Return information about an ADT

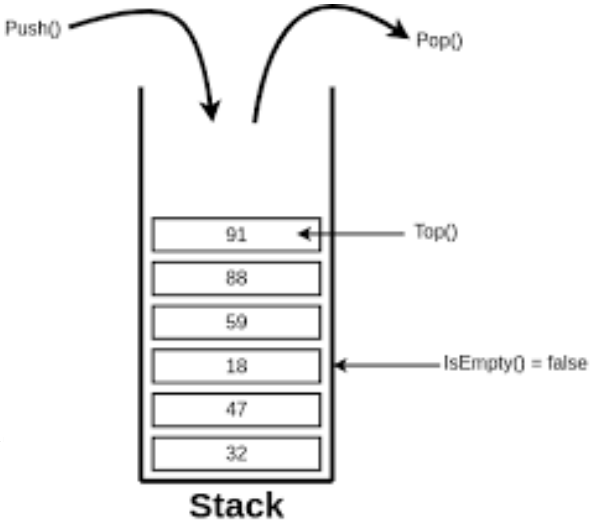
How to implement an ADT

1. Study properties of data abstraction and develop a list of allowable operations
2. For each operation in the list:
   - (a) Derive the required client/object interactions
   - (b) Specify methods for those interactions
3. Code each method maintaining the properties of the data abstraction
4. Test the implementation of each method



Implementation of Stack ADT

- LIFO collection of homogeneous data items, where all insertions and deletions occur at one end (top of the stack). Implementation choice: Array or Linked List



- Allowed operations:
  - Creators: Constructor of java.util.Stack.
  - Producers: Vector(Collection c) method of Vector
  - Observers: peek() and isEmpty() methods of java.util.Stack.
  - Mutators: push(E item) and pop() methods of java.util.Stack
- Available operations on java.util.Stack:
  - push(E e) – Inserts an element at the top of stack.
  - pop() – Removes element from the top of the stack if it is not empty
  - peek() – Returns the top element of stack without removing it.
  - size() – Returns the size of the stack.
  - isEmpty() – Returns true if the stack is empty, else it returns false.
- Alternative implementation example can be with array, interface of methods and programmer-written method implementation

ADT operations: Specification example

```
/** String represents an immutable sequence of characters.
    Convert a boolean value to string: true --> "True", false --> "False" */
public class MyString {
     private char[] str;

    /*--- creator operation ---*/
    /** @param b a boolean value
      * @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }

    /*--- observer operations ---*/
    /** @return number of characters in this string */
    public int length() { ... }

    /** @param i character position (requires 0 <= i < string length)
      * @return character at position i
      */
    public char charAt(int i) { ... }

    /*--- producer operation ---*/
    /** Get the substring between start (inclusive) and end (exclusive).
     * @param start starting index
     * @param end ending index.  Requires 0 <= start <= end <= string length.
     * @return string consisting of charAt(start)...charAt(end-1)
```

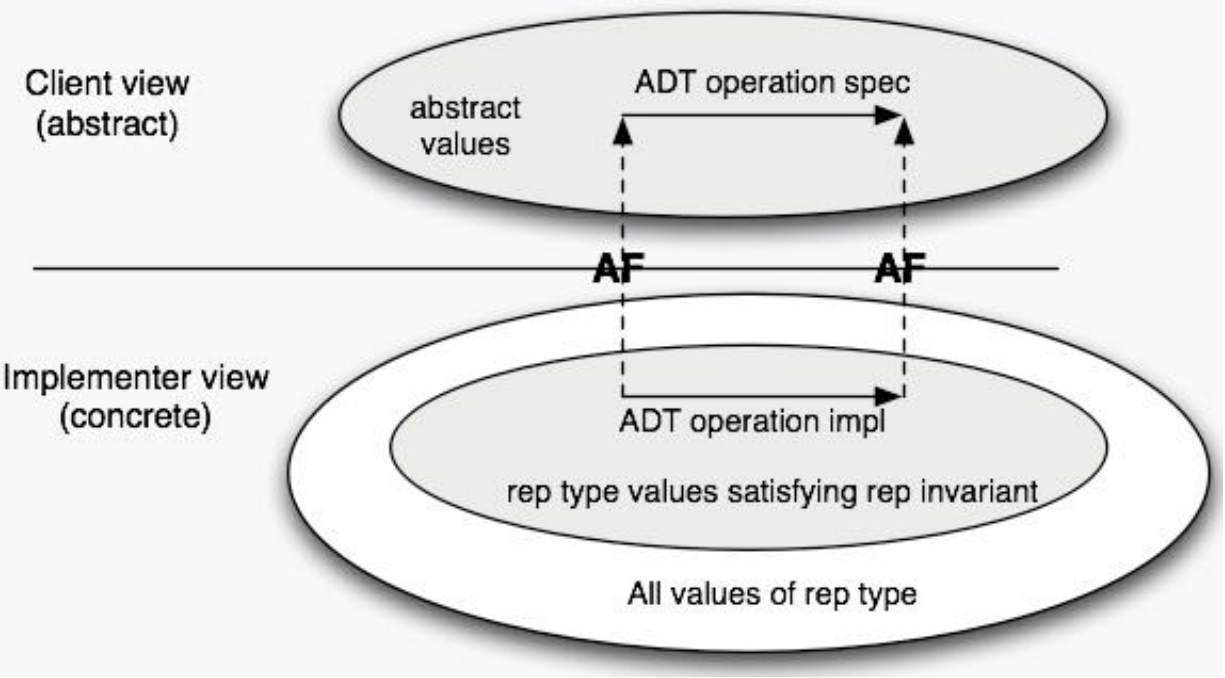
ADT operations: Implementation example

```
/** String represents an immutable sequence of characters. Convert bool to str */
public class MyString {
    private char[] str;
    /*--- creator operation ---*/
    public static MyString valueOf(boolean flag) {
        MyString s1 = new MyString();
        s1.a = flag ? new char[] {'R','i','g','h', 't'} : new char[]
{'W','r','o','n','g'};
        return s1;
    }

    /*--- observer operations ---*/
     public int length() { return str.length; }
     public char charAt(int i) { return str[i]; }

    /*--- producer operation ---*/
    public MyString substring (int start, int end) {
        MyString s1 = new MyString();
        s1.str = new char[end - start];
        System.arraycopy(this.str, start, s1.str, 0, end - start);
        return s1;
    }
}
Test:
MyString s = MyString.valueOf(false); System.out.print(s.charAt(2)); //
assertEquals(4, s.length());
```

Concrete State Space and Rep Invariant

- Concrete State Space: Set of all possible values of the Data Representation. It describes the Implementer View.

- Rep Invariant: Mandatory condition for all valid concrete representations of a class. Defines the domain of the abstraction function, e.g. line requires r.startX ≠ r.endX or r.startY ≠ r.endY



Abstraction Function

- Abstraction Function(AF): A function from an object's concrete representation (R) to the abstract value (A) it represents, AF: R ⇒ A

```
/**
 *  Complex represents an immutable complex number:
 *  Means (A)
 */
public class Complex {
    private double real;            // Part of (R)
    private double imag;            // Part of (R)

    // The abstraction function is
    //     AF(r) = r.real + i * r.imag
    ...
}
```