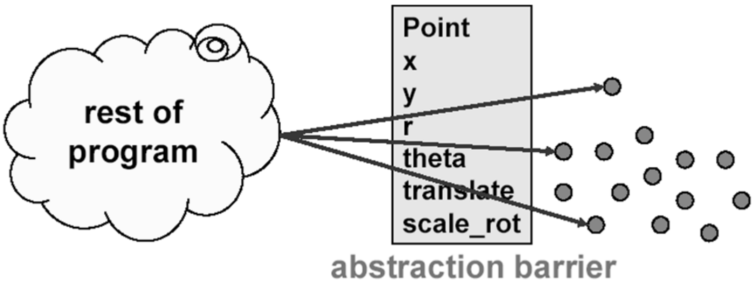
From Data Structures to Abstract Data Types (ADTs)

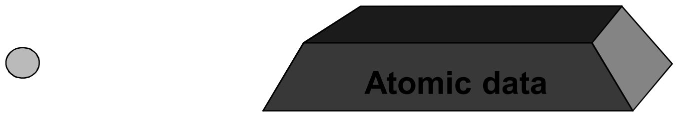


Data Collections

- As our programs become more sophisticated, we need assistance :
  - to organize large amounts of data
  - to manage relationships among individual data items
- Organizing data into collections plays an important role in almost all non-trivial programs
- A collection is a group of individual data items
  - that we want to treat as a conceptual unit
  - while preserving their relationships
- Common types of data collections are:
  - Arrays, Lists, Stacks, Queues, Trees, Graphs, Sets, Bags, Maps, …

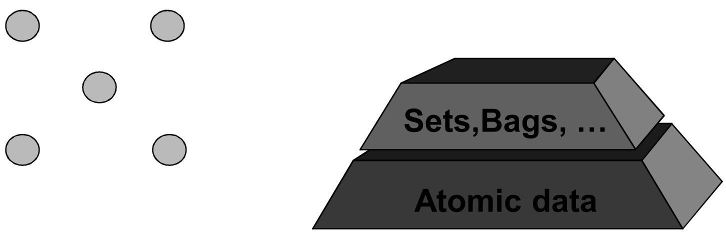
Data Collection Categories

- Individual data items: basic data types



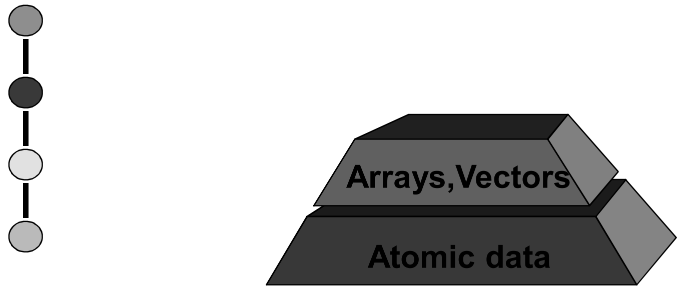
Data Collection Categories

- Unordered data collections: Sets, Bags, Maps (Table)



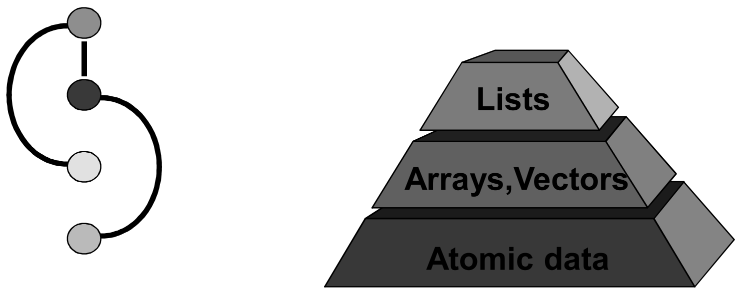
Data Collection Categories

- Ordered data collections: arrays, vectors



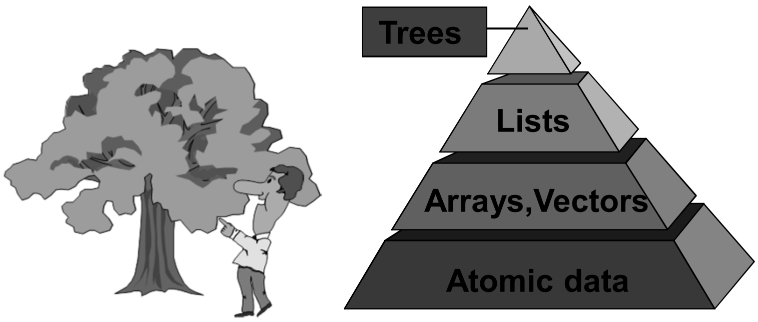
Data Collection Categories

- Linear data collections: lists



Data Collection Categories

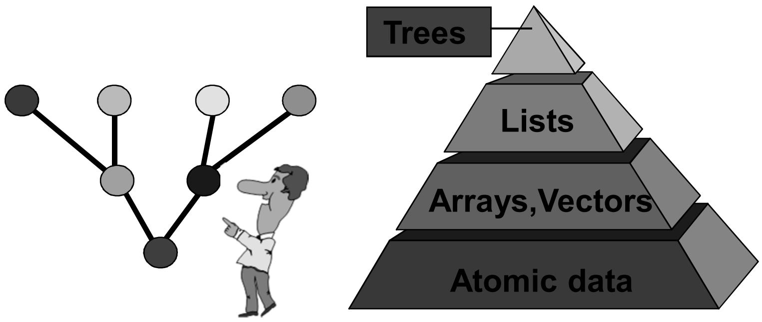
- Hierarchical data collections: Trees



CSD Univ. of Crete
Fall 2012

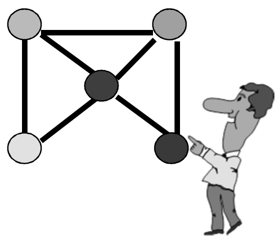
Data Collection Categories

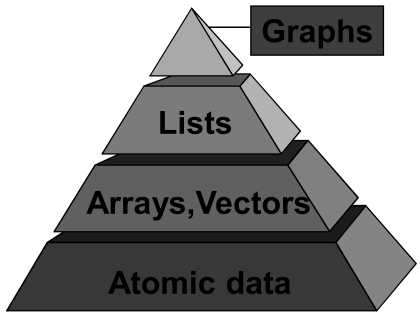
- Hierarchical data collections: Trees



Data Collection Categories

- Graph data collections: Graphs





Some Common Operations

- Search and retrieval
  - Search a collection for a given item or for an item at a given position. Usually return the item or its position, or return some distinguishing value like null or -1 if the item is not found
- Removal
  - Delete a given item or an item at a given position
- Insertion
  - Add an item to a collection, usually at some particular position
- Replacement
  - Combination of removal and insertion

Some Common Operations (Cont.)

- Traversal
  - Visit each item in a collection. Traversals visit items in some specific order. Some traversals allow modification to the collection being traversed
- Test for Equality
  - Test a collection of items for equality. Every item must be an instance of a type that can be tested for equality
- Size of a collection
  - Determine the number of items in a collection. This number is a collections size
- Cloning
  - Make a copy of an entire collection. Each item in the collection needs to also be copied

Arrays: The Most Common Data Collection

- Arrays represent a sequence of data items that can be accessed by index position
  - Each item has a numeric index position
  - Once an array is created, it has a fixed size
- The index operation is very fast and it makes storing and retrieving items from a given position very efficient
  - No matter how large an array, it takes constant time to access the first or the last item
- An array stores multiple values of the same type
  - can be primitive types or objects
  - Therefore, we can create an array of integers, characters etc. or an array of objects of a specific class

Using Java Arrays

- In Java, arrays are "object" or reference types in their own right, regardless of what they store
  - ◆The name of the array is an object reference variable, and the array itself is instantiated separately
  - ◆The type of the array does not specify its size, but each object of that type has a specific size
- Three step process
  - ❶Declare an array variable
  - ❷Create a new array "object" and assign the array to the array variable
  - ❸Store values or objects in the array

Step 1: Declare Array

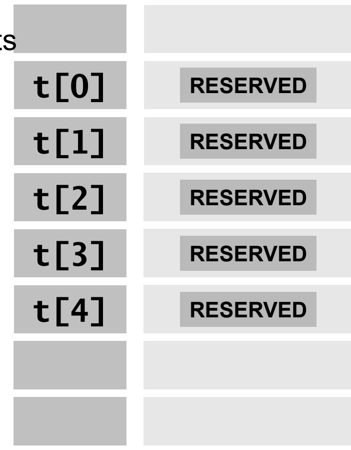
- To declare an array variable you specify:
  - ◆The type of elements you'll store in the array
    - This can be any type, object or primitive
  - ◆A name for the entire collection
    - Following standard naming rules for identifiers
  - ◆A set of empty brackets following either array name or element type
    - Java programmers tend to favor brackets after type
    - Associates brackets with type rather than with variable
- Examples:

```
int vals[];        int[] vals;        int[100] vals;
char test[];       char[] test;       char test[5];
```

Step 2: Create Array

```
test = new char[5];
```

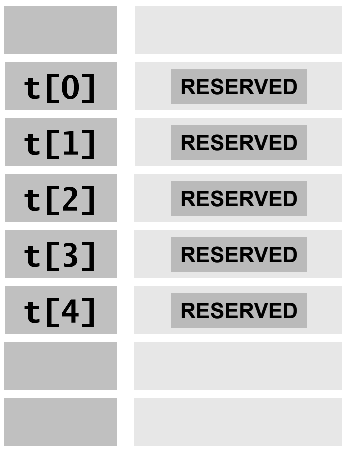
- Array is created with **new** just like other objects
- Special array syntax for **new**:
  - Note the use of brackets `[ ]` rather than parentheses
  - This is an array constructor, not an object constructor
- The elements in a new array have:
  - zero, if they are numeric
  - null, if the elements are objects
- Arrays are indexed starting with zero
  - Arrays must be indexed by `int` values (`short`, `byte`, or `char` are OK, but `long` is no good)



Step 1 + 2: Declare and Create

```
char[] test = new char[5];
```

- We can declare and create an array in one statement
- Elements are numbered from 0 to length-1
- Every array has a public field, `length`, that stores the number of elements in the array
- **`t[2]`** refers to the third element of the array **`test`**
  - The expression represents a place to store a single char, can be used wherever a character variable can

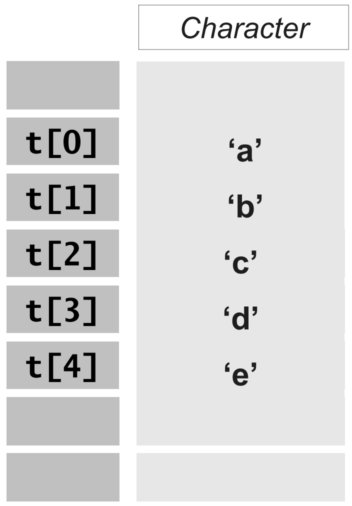


8

Step 3: Store Values

- `t[2]` refers to the third element of the array `test`
  - The expression represents a place to store a single char, can be used wherever a character variable can
- Store (or read) array values using subscripts: `test[ 3 ] = ‘d’;`
- Use a loop to fill in integer values

```
char[] t = new char[5];
int i;
for (i=0;i<5;i++)
  t[i] = (char)('a'+i);
```

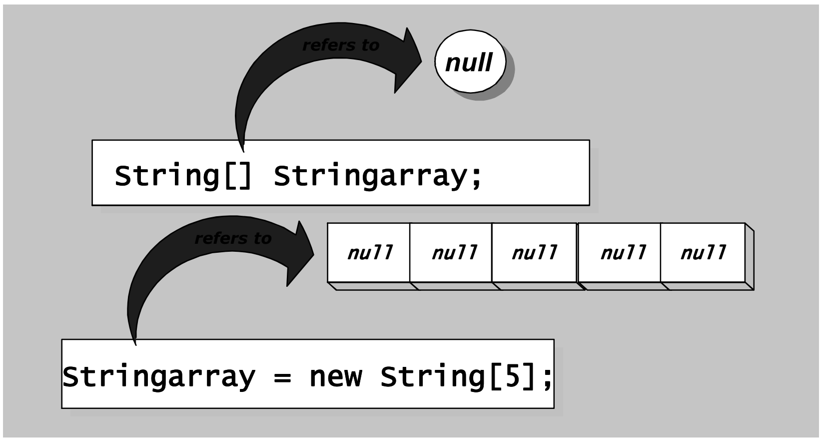


Initializing Arrays

- You can initialize an array by providing a set of values in the declaration
  - the new operator is not used
  - no size value is specified
- The size of the array is determined by the number of items in the initializer list
  - values are delimited by braces and separated by commas

```
char[] hextab = {
   '0','1','2','3','4','5','6','7',
   '8','9','A','B','C','D','E','F'
   };
```

Array of Objects

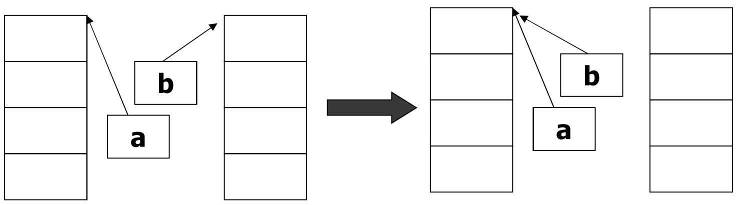


Object Type Array: Remarks

- When you declare an array, you create a reference to an object

```
int [] a = new …
int [] b = new …

b = a;   // doesn't copy a to b!
```



- This is NOT a pointer, thus you can not manipulate it using pointer arithmetic

```
float [] fa = new float[10];
// some code goes here.
// I want to point to the second element
fa++;  // Error !
```

Multidimensional Arrays

- Think of rows, columns and grids
- Declare using multiple brackets

  `int [ ] [ ] ia2= new int[3][5];`
  - `ia2` has 3 rows, 5 columns
- Address each element using two subscripts

  `ia[1][4] = 30`
  - Puts value 30 in last element of second row
  - Remember: arrays are numbered 0 to length-1

Multidimensional Arrays

```
int[ ][ ] scores = new int[3][3];
```

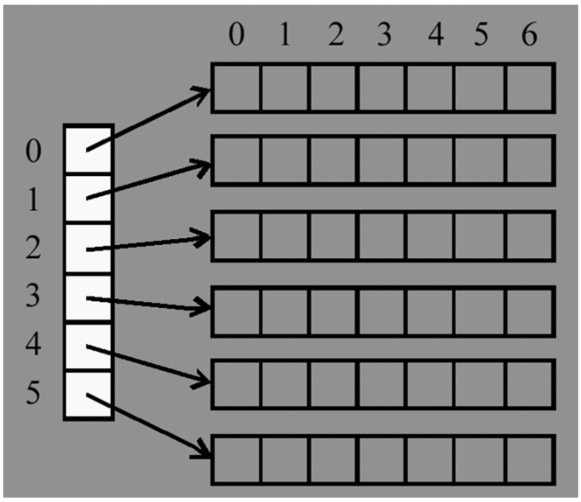
	[0]	[1]	[2]
[0]	50	100	12345
[1]	0	735	89
[2]	12389	7	88

```
scores[0][0] = 50;
scores[0][1] = 100;
scores[0][2] = 12345;
scores[1][0] = 0;
scores[1][1] = 735;
scores[1][2] = 89;
scores[2][0] = 12389;
scores[2][1] = 7;
scores[2][2] = 88;
```


11

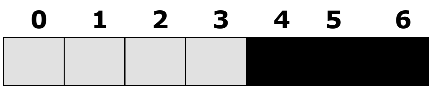
Multidimensional Arrays: The Truth!

- Multidimensional Arrays are in reality Arrays of Arrays !!!



Physical vs. Logical Array Size

- Physical Size: the total number of array cells
  - That is, the number used to specify the capacity when the array was created or resized
  - An array of size N is indexed from zero to N-1
- Logical Size: the number of items that have been added to the array
  - If we want to keep track of the logical size of an array we need to do it ourselves with a counter



Physical Size = 7

Logical Size = 4

Bounds Checking

- In Java, the array itself is an object and has a public constant called `length` that stores the size of the array
  - `length` holds the physical size, not the logical
- Each array object is referenced through the array name (just like any other object):
  - The name of the array is an object reference variable

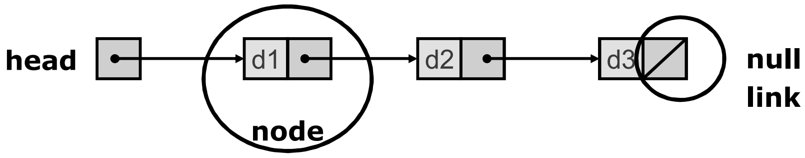
    `my_array.length`
- The Java interpreter will throw an exception if an array index is out of bounds
  - This is called automatic bounds checking

Problems with Array-Based Data Structures

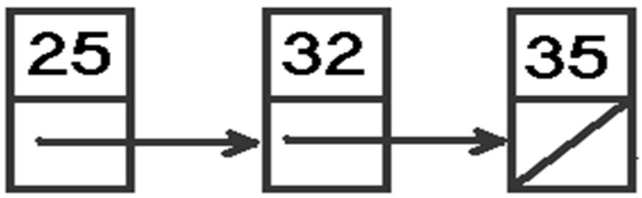
- Insertions and deletions incur some overhead
  - Must shift items to open or close a hole
  - Must copy all items during resizing in a dynamic implementation
- There is a one-to-one correspondence between the logical position of a cell in the array and its physical position in memory
  - Require contiguous memory (cells must be physically adjacent)
- If we could decouple the logical position of a cell from its physical position, we could
  - add cells or remove them at no extra cost
  - perform insertions or removals of data items without shifting existing data items
- Note that Java provides the ***Vector class*** to produce array structures can that can dynamically grow or shrink

Linked Lists: Another Common Data Collection

- Linked lists consists of data items called nodes
  - A node contains data and one or more links to other nodes
- Linked data structures are dynamic
  - memory is allocated for new data items as needed (no need to resize)
  - items are linked to other items through references/pointers
- To access an item of a linked list
  - we access the head and then follow the links to the item we want
  - the last item in a linked structure has no link this is called a null link

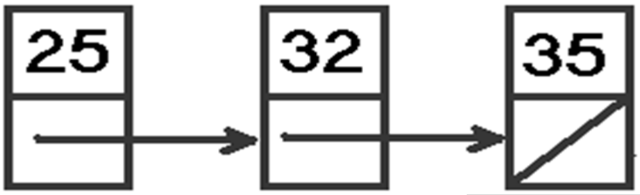


Linked list in C




```
struct my_node
{
   int value;
   struct my_node* next;
}
```

Linked list in JAVA

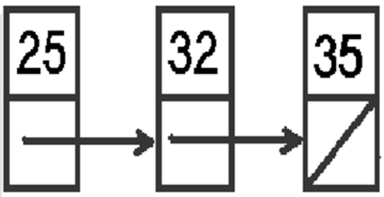


The constructor may be different from this

```
public class my_node
{
   int value;
   my_node  next;
   my_node(int v,  my_node node)
   { … }
   Other methods as needed
}
```

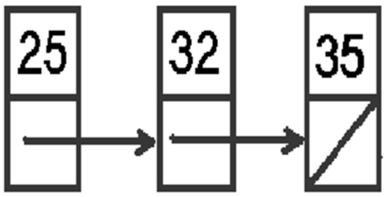

Creating a linked list in C

```
struct my_node *start=NULL,*new;
new=malloc(sizeof(struct my_node));
new->value = 35;
new->next = start;
start = new;
new=malloc(sizeof(struct my_node));
new->value = 32;
new->next = start;
start = new;
new=malloc(sizeof(struct my_node));
new->value = 25;
new->next = start;
start = new;
```



Creating a linked list in Java

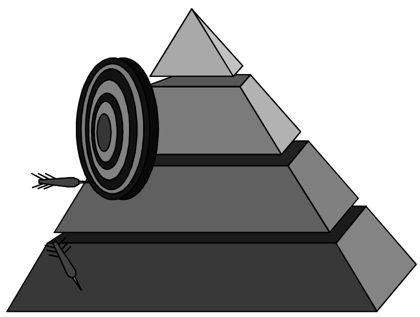
```
my_node start;
start=new my_node(35, start);
start=new my_node(32, start);
start=new my_node(25, start);
```



The process is basically the same!!

From Data Structures …

- Whenever we need to organize data into collections we should consider the pros and cons of memory requirements and complexity of each possible implementation of a collection
- Choosing the most appropriate to our needs data structures and operations to implement collections is as important as the choice of algorithms in program development



… to Data Abstraction …

- Data collections should be better to be modeled as abstractions, hiding as much as possible implementation details
- Clients
  - ◆Interested in WHAT services a module provides, not HOW they are carried out
  - ◆So, ignore details irrelevant to the overall behavior, for clarity
- Implementers
  - ◆Reserve the right to change the code, in order to improve performance
  - ◆So, ensure that clients do not make unwarranted assumptions

… and Abstract Data Types (ADTs)

- An ADT is a programmer-defined type with a set of data values (domain), and a collection of allowable operations on those values
  - ◆The set of Operations define the interface to the ADT
  - ◆Data Structures and Program Code are essentially the physical implementation of an ADT

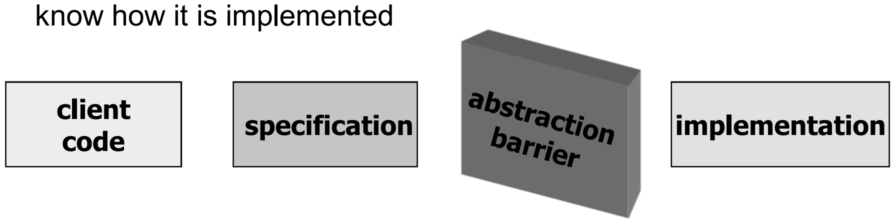
Specification Tasks	**Implementation Tasks**
Describe the domain of ADT **Select and describe ADT operations**	**Choose concrete data representation for ADT** **Code all ADT operations in a PL**

Abstract Data Types

- An ADT defines a concept of what a particular collection of data items is, and a data structure tells us how we are going to represent that concept instances and implement their behavior in our program
  - ◆Data Types: values, operations, and data representation
  - ◆Abstract Data Types: values and operations only
- ADTs are not characterized by their concrete data representation (i.e., structure)
  - ◆The data representation is private, so application code cannot access it: only the operations can
  - ◆The data representation is changeable, with no effect on application code: only the operations must be recoded

Abstract Data Types: Encapsulation of Data

- Data values and code for operations are encapsulated within an abstraction barrier to support 2 benefits of ADTs:
  - ❶ The creator of the ADT guarantees that the user can access the encapsulated data only through the allowable operations
  - ❷ The user is guaranteed the ability to use the ADT without having to know how it is implemented



- Objects are a perfect programming mechanism to create ADTs because their internal details are encapsulated

ADTs and Contract-based Programming

- Each ADT should have a contract that:
  - ◆specifies the set of values of the ADT
  - ◆specifies each operation of the ADT (i.e., the operation's name, parameter type(s), result type, and observable behavior)
- The contract does not specify the data representation, nor the algorithms used to implement the operations
- The observable behavior of an operation is its effect as 'observed' by the client code
  - ◆Example of observable behavior: search an array
  - ◆Examples of algorithms with that behavior: linear search, binary search

ADTs and Contract-based Programming

- The ADT programmer undertakes to provide an implementation of the ADT that respects the contract
  - ◆must choose a concrete data representation using the data types already supported by a PL implement each allowable operation in terms of PL instructions
- The application programmer undertakes to process values of the ADT using only the operations specified in the contract
- Separation of concerns:
  - ◆The ADT programmer is not concerned with what applications the ADT is used for
  - ◆The application programmer is not concerned with how the ADT is implemented
- Separation of concerns is essential for designing and implementing large software systems

ADT Example: The Pushdown Stack Contract

- A stack is a linear data structure with homogeneous data items (elements), in which all insertions and deletions occur at one end, called the top of the stack.
  - A stack is a LIFO “Last In, First Out” structure
- Stacks are managed using mainly two functions:
  PUSH - places an element on top of the stack
  POP - removes an element from the stack
- Analogy: a stack of plates
- Java has a built-in ***Stack class*** that extends the ***Vector class***

ADT Stack Operations

- Constructors
  - **Init**: creates an empty stack
- Transformers
  - **Push**: adds a new item to the top of the stack
  - **Pop**: removes the item at the top of the stack

change state

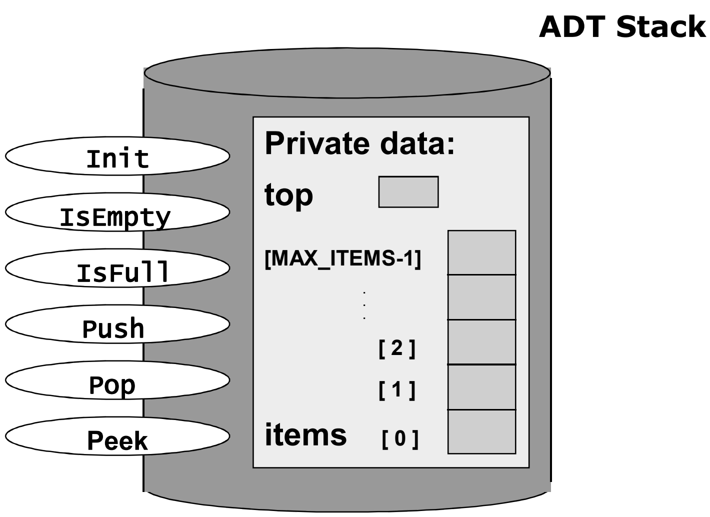
- Observers
  - **IsEmpty**: determines whether the stack is currently empty
  - **IsFull**: determines whether the stack is currently full
- Accessors
  - **Peek** : returns a copy of the item currently at top of the stack

observe state

ADT Stack Implementation

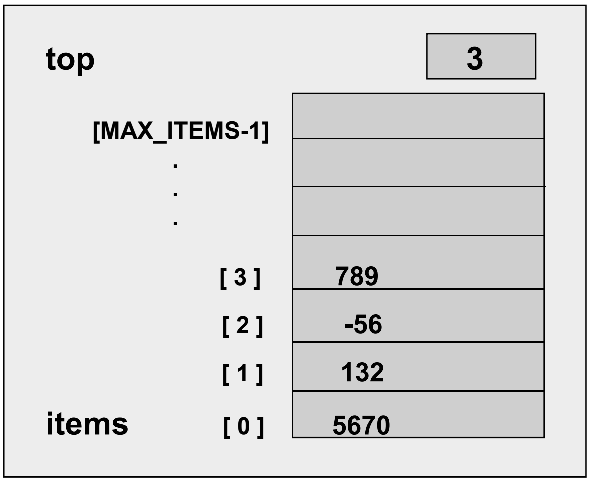
- As long as the ADT accurately fulfills the promises of its contract, it doesn't really matter how the ADT is implemented
- We can change the ADT implementation without affecting client programs using the ADT interface
- An implementation of an ADT entails:
  - choosing a data representation
  - choosing an algorithm for each operation
- The data representation must be private and cover all possible values
- The algorithms must be consistent with the data representation
- Two possible Stack Implementations:
  - Using Arrays: the maximum size of the stack is fixed at compile time
  - Using Linked Lists: we can dynamically allocate the space for each stack element as it is pushed onto the stack

Implementing the ADT Stack using Arrays

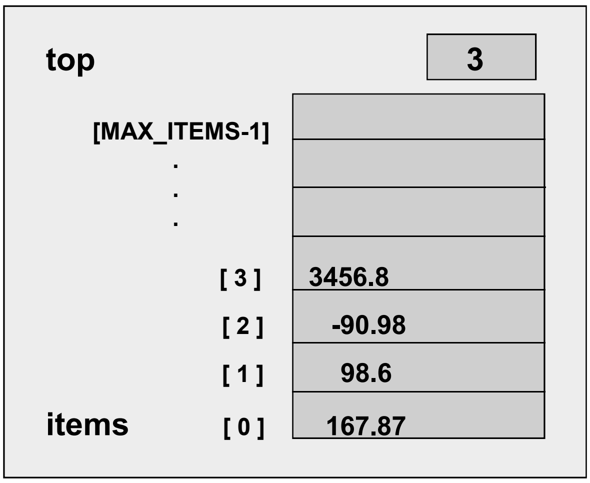


21

Stack of Integer Items



Stack of Float Items



Tracing Client Code

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code

letter	'V'

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

Tracing Client Code

letter 'V'

Private data:

top -1

[MAX_ITEMS-1]

.
.
.

[2]

[1]

items [0]

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

47

Tracing Client Code

letter 'V'

Private data:

top 0

[MAX_ITEMS-1]

.
.
.

[2]

[1]

items [0] 'V'

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

48

Tracing Client Code

letter 'V'

Private data:

top 1

[MAX_ITEMS-1]	
. . .	
[2]	
[1]	'C'
items [0]	'V'

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code

letter 'V'

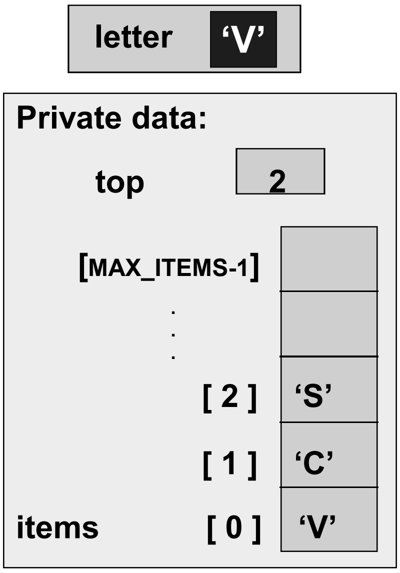
Private data:

top 2

[MAX_ITEMS-1]	
. . .	
[2]	'S'
[1]	'C'
items [0]	'V'

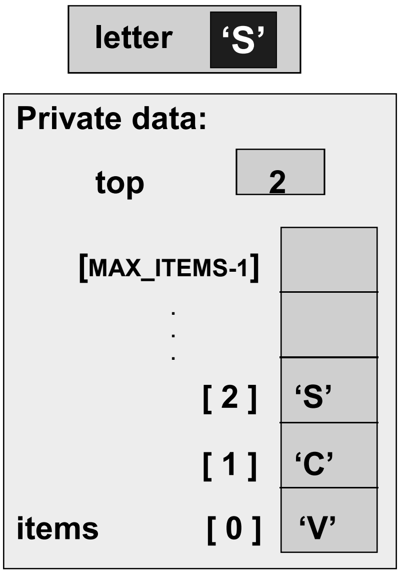
```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

Tracing Client Code




```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
  letter = charStack.Peek( );
  charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code




```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
  letter = charStack.Peek( );
  charStack.Pop( );
}
charStack.Push('K');
```

Tracing Client Code

letter 'S'

Private data:

top 1

[MAX_ITEMS-1]

.
.
.

[2] 'S'

[1] 'C'

items [0] 'V'

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code

letter 'S'

Private data:

top 2

[MAX_ITEMS-1]

.
.
.

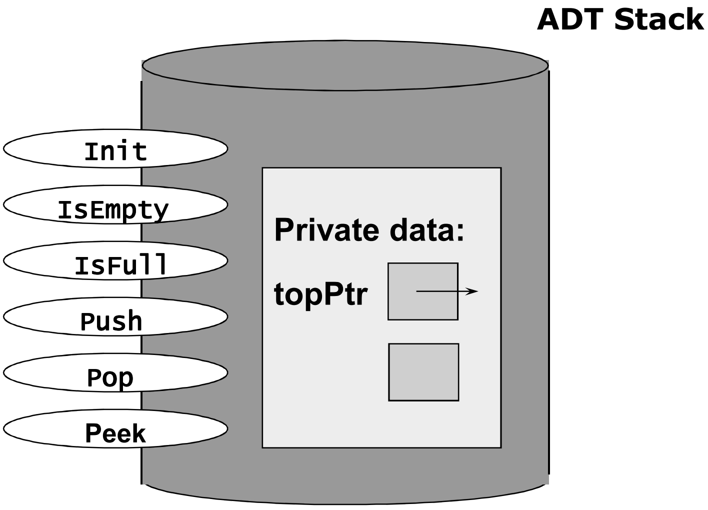
[2] 'K'

[1] 'C'

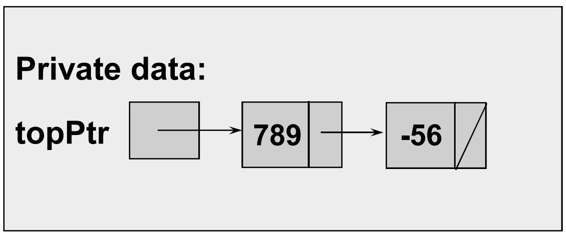
items [0] 'V'

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

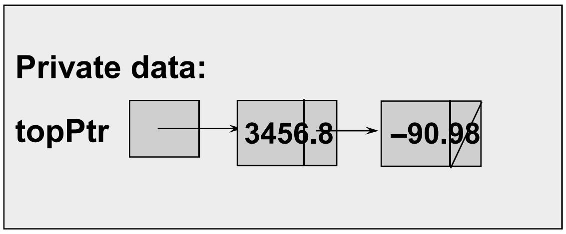
Implementing the ADT Stack using Linked Lists



A Stack of Integer Items



A Stack of Float Items

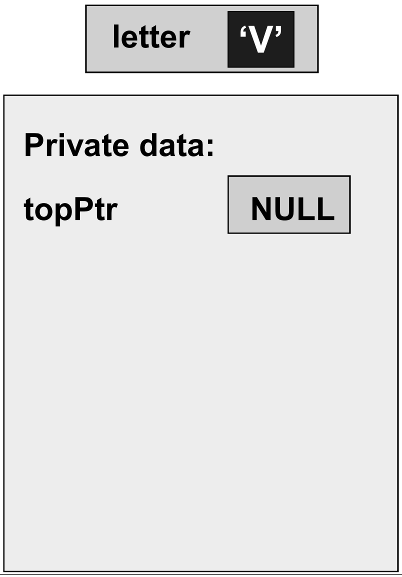


Tracing Client Code

letter ‘V’

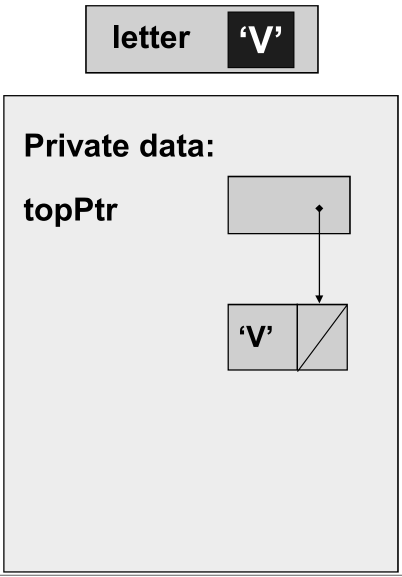
```
char letter = ‘V’;
Init charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push(‘K’);
```

Tracing Client Code



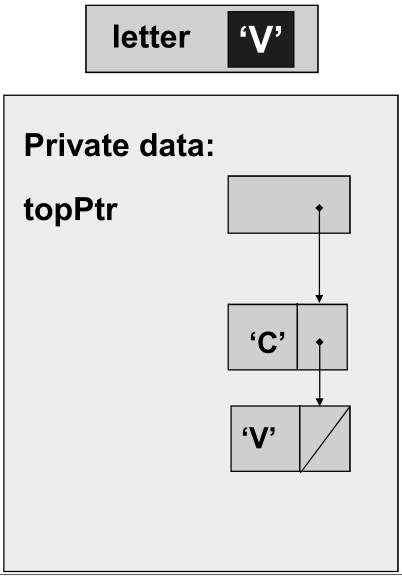

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code



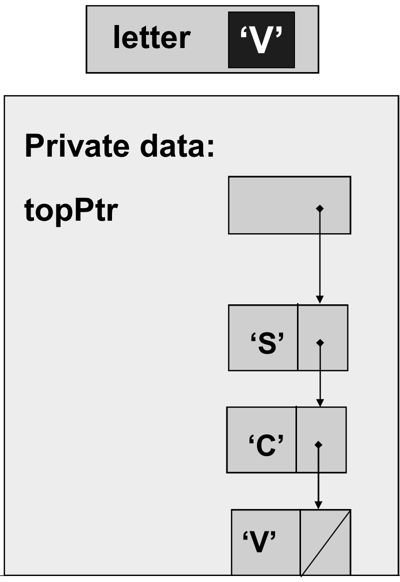

```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )){
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

Tracing Client Code




```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

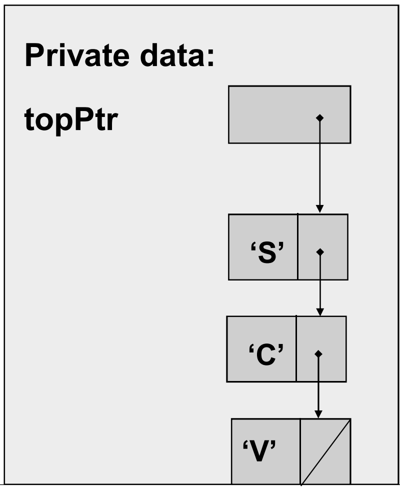

Tracing Client Code




```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

Tracing Client Code

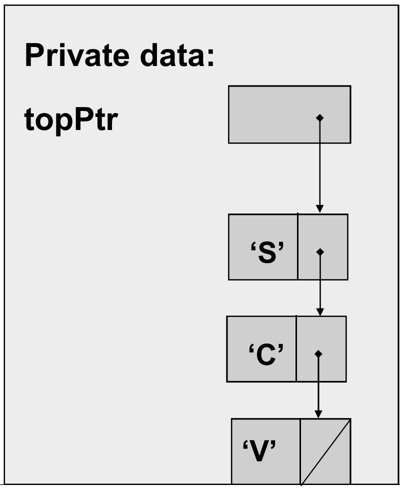
letter ‘V’




```
char letter = ‘V’;
Init charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push(‘K’);
```


Tracing Client Code

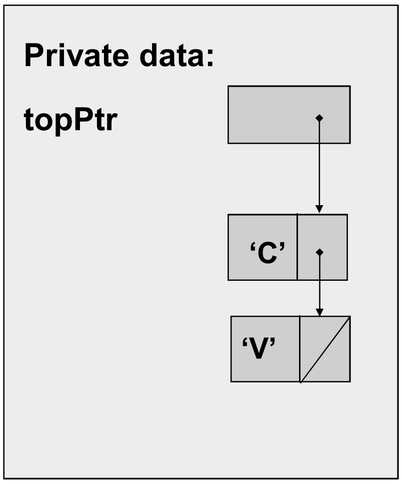
letter ‘S’




```
char letter = ‘V’;
Init charStack;
charStack.Push(letter);
charStack.Push(‘C’);
charStack.Push(‘S’);
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push(‘K’);
```

Tracing Client Code

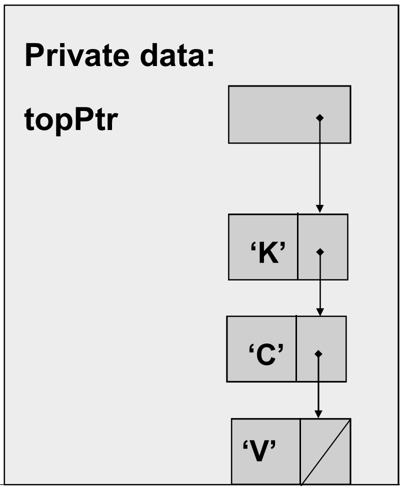
letter **'S'**




```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```


Tracing Client Code

letter **'S'**

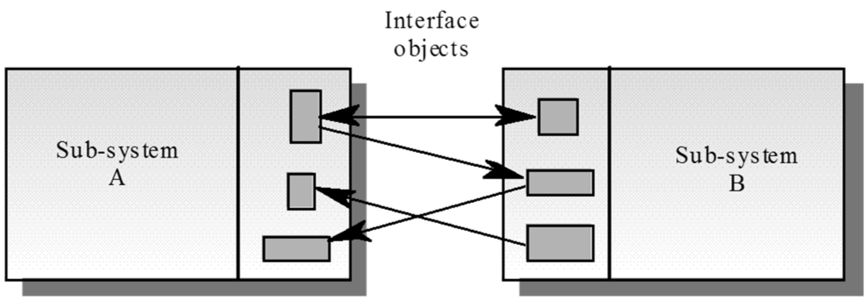



```
char letter = 'V';
Init charStack;
charStack.Push(letter);
charStack.Push('C');
charStack.Push('S');
if (!charStack.IsEmpty( )) {
 letter = charStack.Peek( );
 charStack.Pop( );
}
charStack.Push('K');
```

Recall Modularity

- We divide the system into several autonomous components (objects)
  - Each has a well defined role and interface
  - These components interact between them to achieve the functionality of the whole system
- The components themselves can be further divided into smaller components
- Building our system from smaller components, has several advantages:
  - It is easier to understand the systems in terms of a collection of few interoperating components
  - We can correct/improve the implementation of one component, without affecting the others
  - A component may be used later in other places

Interface Specification



- Large systems are decomposed into modules with well-defined interfaces between these subsystems
- Specification of module interfaces allows independent development of the different modules
- Interfaces may be defined as abstract data types (ADT) or object classes

Design by Contract: Basic Notions

- Parties in the contract:
  - ADT supplier and clients
- Design by contract:
  - Relationship between ADT supplier and clients is a formal agreement
- The presence of a precondition (input requirement) or postcondition (output requirement) in an ADT operation is viewed as a contract:
  - If you promise to call an ADT operation with `pre` satisfied then I, in return, promise to deliver a final state in which `post` is satisfied
- Contract:
  - entails benefits and obligations for both parties

Pre- and Post Conditions

- Preconditions: client's promise to the operation
  - expresses the constraints under which an operation will function properly
- Postconditions: supplier's promise to the operation
  - expresses properties of the state resulting from an operation's execution
- Precondition binds the client
  - It is an obligation for the client and a benefit for the supplier
- Postcondition binds the supplier
  - It is an obligation for the supplier and a benefit for the client

Contract Benefits and Obligations

	Obligations	Benefits
Client	Satisfy precondition: Only call `push(x)` if the stack is not full	From postcondition: Ensure that stack gets updated to be non empty, with x on top
Supplier	Satisfy postcondition: Updated representation to have x on top, not empty	From precondition: No need to treat cases in which the stack is already full

What happens If a Precondition is Not Satisfied?

- If client's part of the contract is not fulfilled, supplier can do what it pleases:
  - return any value, loop indefinitely, terminate in some wild way!
- Advantage of the convention: simplifies significantly the programming style
  - Does data passed to a method satisfy requirement for correct processing?
    - Problem: no checking at all or multiple checking
    - Multiple checking: Due to redundancy it complicates code maintenance
    - Recommended approach:
      - use preconditions !!!

Assertions

- An assertion is a programmer's claim (with a value of true or false) about the contents of program variables at a particular location in program execution
  - In theory, assertions are first-order logic formulae
  - In a programming language, assertions are computable boolean expressions that can contain program variables, arithmetic/boolean operations, and possibly, user-defined functions
- **Pre** and **Post** conditions are a pair of assertions used to document the behavior of an ADT
  - In general, the preconditions must not use features hidden from the clients
  - However, the postconditions can use any feature, even though only clauses without hidden features are directly usable by the client

Invariants

- Preconditions and postconditions describe the properties of individual operations
- There is also a need for expressing global properties of the instances of an ADT, which must be preserved by all operations
  - Such properties will make out the ADT invariants
- Examples
  - `0<=nb_elements; nb_elements<=max_size`
  - `empty=(nb_elements==0);`
- Must be satisfied by all instances of the ADT at all "stable" times (state):
  - on instance creation
  - before and after every call to an operation (may be violated during call)
- An invariant applies to all contracts between an operation of the ADT and a client
  - acts as control on the evolution of type instances

Subcontracting: What Inheritance is About?

- Subcontractor must do job originally requested:
  - Could do less by
    - requesting a stronger precondition
    - ensuring a weaker postcondition
  - Could do more by
    - accepting weaker precondition
    - guaranteeing a stronger postcondition

Design by Contract: Advantages

- Ensure the correctness of our software:
  - Reliability (Assertions)
- Recover when it is not correct anyway:
  - Robustness (Exception handling)
- Aid in documentation
- Aid in debugging

- Design by Contract:
  - Pre-Post conditions: Rights and Obligations
  - Exceptions: Contract Violations

- Example: Ariane 5 crash, $500 million loss
  - Conversion from a 64 bit # to 16 bit
  - The number didn't fit in 16 bits
  - Analysis had previously shown it would, so monitoring that assertion was turned off

But How to Prove Correctness?

- A complex story: Verifiable Programming
- Reason about imperative sequential programs
- Imperative program defines
  - state space
    - defined by collection of typed variables programs
    - are coordinate axis of state space
  - pattern of actions operating in state space

Formal Methods

- Formal specification consists of techniques for the unambiguous specification of software
- Formal specification is part of a more general collection of techniques that are known as ‘formal methods’
- These are all based on mathematical representation and analysis of software
- Formal methods include
  - Formal specification
  - Specification analysis and proof
  - Transformational development
  - Program verification

Acceptance of Formal Methods

- Formal methods have not become mainstream software development techniques as was once predicted
  - Other software engineering techniques have been successful at increasing system quality
    - Hence the need for formal methods has been reduced
  - Market changes have made time-to-market rather than software with a low error count the key factor
    - Formal methods do not reduce time-to-market
  - The scope of formal methods is limited
    - They are not well-suited to specifying and analyzing user interfaces and user interaction
  - Formal methods are hard to scale up to large systems
- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems
  - In this area, the use of formal methods is most likely to be cost-effective

Formal Specification of Software Systems

- A model-based specification of a system is given in terms of a state model that is constructed using mathematical constructs such as sets sequences, trees, maps, etc.
  - operations are defined by modifications to the system's state
- An algebraic specification of a system is given in terms of its operations and their relationships
  - captures the least common-denominator (*behavior*) of all possible implementations
  - the algebraic specification is well-suited to interface specification

	Sequential	Concurrent
Alge braic	Larch (Guttag, Horning et al., 1985; Guttag, Horning et al., 1993), OBJ (Futatsugi, Goguen et al., 1985)	Lotos (Bolognesi and Brinksma, 1987),
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

Towards a Formal Specification of ADTs

ADT : *Values + Operations*

Specify

Syntax — Signature of Ops

Semantics — Meaning of Ops

Model-based: Description in terms of standard "primitive" data types

Axiomatic (Algebraic): Give axioms satisfied by the operations

Parts of an ADT Algebraic Specification

- Introduction
  - Defines the sort (the ADT name) and declares other type specifications that are used
- Description
  - Gives an informal description of the operations on the ADT
- Signature
  - Defines the syntax of the ADT operations in the interface and their domains as well as input and output
- Axioms
  - Defines the operation semantics in terms of equational axioms, that describe their behavioral properties

Informal Specification Example: the ADT Stack

- Intuitively:
  - **init** : creates a new (empty) stack
  - **push** : adds a new item to the top of the stack
  - **peek** : returns a copy of the item on the top of the stack
  - **pop** : removes the top item
  - **isempty** : tests for an empty stack
- Basic assumptions
  - no stack overflow
- Axioms in English
  - ❶ a new stack is empty
  - ❷ a stack is not empty immediately after pushing an item onto it
  - ❸ attempting to pop a new stack has no effect
  - ❹ pushing an item onto a stack and immediately popping it off leaves the stack unchanged
  - ❺ there is no top item returned by peek in a new stack (nil)
  - ❻ pushing an item onto a stack and immediately peeking the top item returns the item just pushed onto the stack

Formal Specification Example: the ADT Stack

Example 1 (Algebraic Specification of the Class of Integer Stacks).

module $INTEGER\text{-}STACK$
include $INTEGER$
class $Stack$
imported classes $Integer\ Boolean$
operations
  $init : \rightarrow Stack$
  $\_.isEmpty : Stack \rightarrow Boolean$
  $\_.push(\_) : Stack\ Integer \rightarrow Stack$
  $\_.pop : Stack \rightarrow Stack$
  $\_.peek : Stack \rightarrow Integer \cup \{nil\}$
variables
  $S : Stack$
  $N : Integer$
axioms
  $a_1: init.isEmpty = true$
  $a_2: S.push(N).isEmpty = false$
  $a_3: init.pop = init$
  $a_4: S.push(N).pop = S$
  $a_5: peek = nil\ if\ S.isEmpty$
  $a_6: S.push(N).peek = N$

End of Example 1

42

Characteristics of "Good" ADT Specifications

- Simplicity: Avoid needless features
  - The smaller the interface the easier it is to use the ADT
- 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 ADT interface
- Reusability: Do not customize ADTs to specific clients, but make them general enough to be reusable in other contexts
- Convenience: Where appropriate, provide additional operations (e.g., beyond the complete primitive set) for the convenience of users of the ADT
  - Add convenience operations only for frequently used combinations after careful study

The Syntax of an ADT

- An ADT is defined syntactically by its name and the signature of its operations (for creation, access, etc.)
- ADT Example:
  - name: `Table`
  - operations: `init, size, capacity, lookUp, insert, update, remove, retrieve`
  - signatures:

```
init:      Int -> Table
size:      Table -> Int
capacity:  Table -> Int
lookUp:    Key x Table -> Boolean
insert:    Key x Info x Table -> Table
update:    Key x Info x Table -> Table
remove:    Key x Table -> Table
retrieve:  Key x Table -> Info
```

Terms and Normal Forms

- A term is a composition of operations in an algebraic specification
  - A term essentially records the detailed *history of construction* of the value
  - `retrieve(K,insert(K, I, init(5)))`
- Signatures tells us how to form complex terms from primitive ones
  - *Legal compositions*
    `retrieve(K,insert(K, I, T))` ✓
  - *Illegal compositions*
    `retrieve(insert(K, I, T))` ✗
- A term is in normal form iff it cannot be further transformed by any axiom
  - `retrieve(K,insert(K, I, init(5)))` ✓
  - `remove(K,insert(K, I, init(5)))` ✗
  - Why? `remove(K,insert(K, I, T)) = T`

Equivalent Terms, Ground Terms

- Two terms are said to be equivalent if and only if they can both be transformed to the same normal form
  - `remove(K,insert(K, I, init(5)))`
  - `init(5)`

  are equivalent, because both can be transformed to the normal form
- A term without variables is called a ground term

The Semantics of an ADT

- ADT operations (procedures) are not just pieces of code, they should perform some useful tasks
  - You may specify these tasks by two assertions associated with the operation: precondition and postcondition

- The purpose of a specification is to define the behavior of an ADT
  - Users will rely on this behavior, while implementers must provide it

- Implementation of an ADT is correct relative to a specification
  `{Pre+Invariants} OperationBody {Post+Invariants}`
  - The ADT invariant is implicitly added (anded) to both the precondition and postcondition of every operation of its contract

Pre- and Post Conditions: Example

```
insert :   Key x Info x Table -> Table
           -- Insert an element into a Table
           -- giving its key and related Information.
      require
           -- a valid Key
           key >= 0
           -- the Table has space for another
           -- record
           size( ) < capacity( )
      do
           . . .
      ensure
           -- If the table already had a record with
           -- a key equal to key, then that record is
           -- replaced by entry. Otherwise, entry has
           -- been added as a new record of the Table.
      end
```

precondition

postcondition

Invariant Conditions: Example

```
TABLE creation
 . . .
feature
 . . .
invariant
      size_non_negative: 0<= size( )
      size_bounded: size <= capacity( )
      . . .
end
```


Algebraic Specification Axioms

- Write equational axioms that characterize the meaning of all operations
  - *E.g., identity*, *associativity*, *commutativity* rules
- Constructors: Write *identity* axioms to ensure that two constructor terms that represent the same value can be proven so
- Accessors: Define the meaning of an accessor on all constructor terms, checking for consistency using preconditions
  - **isempty**(**init**(n)) = **true**
  - **size**(**init**(n)) = 0, **capacity**(**init**(n)) = n
- Transformers : Define the meaning of a transformer on all constructor terms, provide *associativity*, *commutativity* axioms
  - **remove**(K,**insert**(K, I, T)) = T
  - **insert**(K, I, T) = if **lookUp**(K, T) then **update**(K,I,T) else **insert**(K, I, T)
  - **retrieve** (K, T) = if **lookUp**(K, T) then **retrieve**(K,T) else *null*
  - **retrieve**(K,**update**(Ki, I, T)) = if K = Ki then I else **retrieve**(K,T)

Completeness and Consistency/Soundness

- Completeness (*No undefinedness*): provide enough operations to build every possible value of the ADT domain
  - Constructors: required for representing values in the domain of the type e.g., **`init`**
  - Accessors: use a value of the ADT to compute a value of some other type e.g., **`isempty`**, **`size`**, **`capacity`**, **`found`**, **`lookUp`**
  - Transformers: compute a new value of the same ADT e.g., **`remove`**, **`insert`**, **`update`**

- Consistency/Soundness (*No conflicts*): provide enough test operations for the client to check all preconditions of the ADT operations
  - `isempty(init(n)) = true`
  - `isempty(insert(K, I, T)) = false`
  - `size(init(n)) = 0, capacity(init(n)) = n`
  - `insert(K,I,T) requires K >= 0 and size(T)<capacity(T)`

Algebraic Specification Example: The ADT GStack

```
init:                 -> GStack
push:    Gstack x G -> GStack
pop:     Gstack      -> GStack
peek:    GStack      -> G
isempty:Gstack      -> boolean
```

```
constructors:   init
transformers:   pop, push
accessors:      peek
observers:      isempty
```

Algebraic Specification Example: The ADT GStack

- *Forall* `s ε GStack Terms x ε G:`

```
pop(push(s,x)) = s

peek(push(s,x)) = x

isempty(init())  = true
isempty(push(s,x)) = false
```

- *Preconditions:*

`pop(s)` *requires* `!isempty(s)`
`peek(s)` *requires* `!isempty(s)`

The ADT (Bounded) GStack

```
init:    int        -> GStack
push:    Gstack x G -> GStack
pop:     Gstack     -> GStack
peek:    GStack     -> G
isempty:Gstack      -> boolean
isfull: Gstack      -> boolean
```

constructors: `init`
transformers: `pop, push`
accessors: `peek`
observers: `isempty, isfull`

- *Auxiliary functions:*

```
Forall s ε Gstack Terms n ε int, n>0, x ε G:

     size(init(n)) = 0
     size(push(s,x)) = 1 + size(s)

     capacity(init(n)) = n
     capacity(push(s,x)) = capacity(s)
```


- *Preconditions:*

```
Forall s ε Gstack Terms, n ε int, n>0, x ε G:

     pop(s)    requires !isempty(s)

     peek(s)   requires !isempty(s)

     push(s,x) requires capacity(s) >= size(s) + 1
```

- *Preconditions:*

```
pop(init(n)) = undefined
peek(init(n)) = undefined
push(x,init(0)) cannot be formed!!

push(x,s) requires   aux(s,1)

     where Forall s ε Gstack Terms:
                          n,m ε int, n>0, x ε G:
          aux(init(n),m) = (m =< n)
          aux(push(x,s),m) = aux(s, m+1)
```



```
Forall  s ε GStack,  n ε int, n>0, x ε G:

     pop(push(s,x)) = s

     peek(push(s,x)) = x

     isempty(init(n)) = true
     isempty(push(s,x)) = false

     isfull(s) = (capacity(s) == size(s))
```

```
isfull(init(n)) = (n == 0)
isfull(push(x,s)) = aux(s,1)

where Forall s ε Gstack Terms:
                 n,m ε int, n>0, x ε G:

     aux(init(n),m) = (m == n)
     aux(push(x,s),m) = aux(s,m+1)
```


Correctness of an ADT

- ADT `T`
- `INV` ADT invariant
- operation `r`: $pre_r(x_r)$ precondition; $post_r$ postcondition
- x_r: possible arguments of `r`
- B_r: body of operation `r`
- $Default_T$: attributes have default values

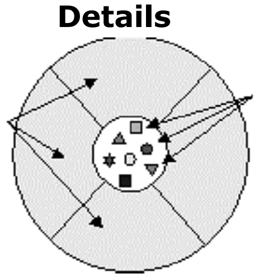
- `T` is said to be correct with respect to its assertions if and only if
  - For every operation `r` other than the constructor (`Init`) and any set of valid arguments x_r:
    {INV and $pre_r(x_r)$} B_r {INV and $post_r$}
  - For any valid set of arguments x_{Init} to the constructor:
    {$Default_T$ and $pre_{Init}(x_{Init})$} B_{Init} {INV}

Invariant Rule

- An assertion I is a correct invariant for an ADT T iff the following two conditions hold:
  - ◆The constructor of T, when applied to arguments satisfying the constructor's precondition in a state where the attributes have their default values, yields a state satisfying I
  - ◆Every public method of the ADT, when applied to arguments and a state satisfying both I and the method's precondition, yields a state satisfying I
- Note that:
  - ◆Preconditions of an operation may involve the initial state and the arguments
  - ◆Postconditions of a method may only involve the final state, the initial state (through old) and in the case of a function, the returned value
  - ◆The ADT invariant may only involve the state

Implementation of an ADT

- The implementation of a type is an interpretation of the operations of the ADT that satisfies all the axioms
- Correctness of a client program is assured even when the implementation is changed
  - ◆ Array-based
  - ◆ LinearList-based
  - ◆ Tree-based
    - • Binary Search Trees, AVL Trees, B-Trees etc
  - ◆ HashTable-based



- These exhibit a common **Stack** behavior, but differ in performance aspects

Object-Based PLs: The Paradigm

- Key feature is Abstract Data Types
  - Supports modularity principles
- Provide encapsulation mechanism for ADT's
  - for grouping data and procedures associated with that data
  - limit outside access to objects inside ADT
  - Examples: Ada packages, CLU clusters, Modula2 modules
- Encapsulating mechanisms themselves tend to be typeless
- Export control mechanisms for types, variables, func/procs in ADT's
- Sometimes import control as well (Euclid)
- Encapsulated ADT's tend to be separately compilable
  - Tends to support programming in the large

Object-Based: History