Abstract Data Types (ADTs), Stacks, Queues

EECS2011 X:
Fundamentals of Data Structures
Winter 2023

CHEN-WEI WANG

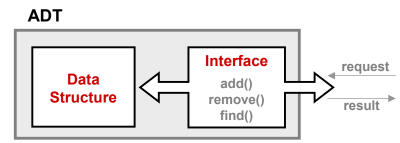
Learning Outcomes of this Lecture

This module is designed to help you learn about:

- The notion of *Abstract Data Types* (***ADTs***)
- ***ADTs***: Stack vs. Queue
- Implementing Stack and Queue in Java [interface, classes]
- Applications of Stacks vs. Queues
- Optional (but highly **encouraged**):
  - Criterion of ***Modularity***, Modular Design
  - ***Circular*** Arrays
  - ***Dynamic*** Arrays, ***Amortized*** Analysis

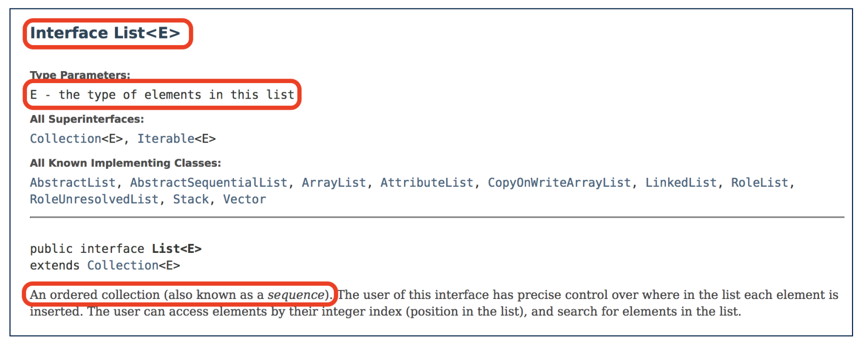
Abstract Data Types (ADTs)

- Given a problem, decompose its solution into *modules*.
- Each *module* implements an *abstract data type (ADT)*:
  - filters out ***irrelevant*** details
  - contains a list of declared ***data*** and well-specified ***operations***



- Supplier's ***Obligations***:
  - Implement all operations
  - Choose the "**right**" data structure [e.g., arrays vs. SLL vs. DLL]
  - The internal details of an implemented *ADT* should be **hidden**.
- Client's ***Benefits***:
  - ***Correct*** output
  - ***Efficient*** performance

Java API Approximates ADTs (1)



Interface List<E>

Type Parameters:
E - the type of elements in this list

All Superinterfaces:
Collection<E>, Iterable<E>

All Known Implementing Classes:
AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as E.
- A reasonably ***intuitive overview*** of the ADT.

Java 8 List API

Java API Approximates ADTs (2)

```
E    set(int index, E element)
     Replaces the element at the specified position in this list with the specified element (optional operation).
```

```
set

E set(int index,
      E element)

Replaces the element at the specified position in this list with the specified element (optional operation).
Parameters:
index - index of the element to replace
element - element to be stored at the specified position
Returns:
the element previously at the specified position
Throws:
UnsupportedOperationException - if the set operation is not supported by this list
ClassCastException - if the class of the specified element prevents it from being added to this list
NullPointerException - if the specified element is null and this list does not permit null elements
IllegalArgumentException - if some property of the specified element prevents it from being added to this list
IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())
```

Methods described in a ***natural language*** can be ***ambiguous***.

Building ADTs for Reusability

- *ADTs* are ***reusable software components*** that are common for solving many real-world problems.
  e.g., Stacks, Queues, Lists, Tables, Trees, Graphs
- An *ADT*, once thoroughly tested, can be reused by:
  - ***Clients*** of Applications
  - ***Suppliers*** of other ADTs
- As a supplier, you are obliged to:
  - ***Implement*** standard ADTs [≈ lego building bricks]
    Note. Recall the basic data structures: arrays vs. SLLs vs. DLLs
  - ***Design*** algorithms using standard ADTs [≈ lego houses, ships]
- For each standard *ADT*, you should know its *interface*:
  - Stored ***data***
  - For each ***operation*** manipulating the stored data
    - How are ***clients*** supposed to use the method? [*preconditions*]
    - What are the services provided by ***suppliers***? [*postconditions*]
    - Time (and sometimes space) ***complexity***

What is a Stack?

- A ***stack*** is a collection of objects.
- Objects in a ***stack*** are inserted and removed according to the *last-in, first-out (**LIFO**)* principle.
  - ***Cannot*** access arbitrary elements of a stack
  - ***Can*** only access or remove the *most-recently added* element

The Stack ADT

- ***top***
  [***precondition***: stack is not empty]
  [***postcondition***: return item **last** pushed to the stack]
- ***size***
  [***precondition***: **none**]
  [***postcondition***: return number of items pushed to the stack]
- ***isEmpty***
  [***precondition***: **none**]
  [***postcondition***: return whether there is no item in the stack]
- ***push(item)***
  [***precondition***: stack is not full]
  [***postcondition***: push the input item onto the top of the stack]
- ***pop***
  [***precondition***: stack is not empty]
  [***postcondition***: remove and return the top of stack]

Stack: Illustration

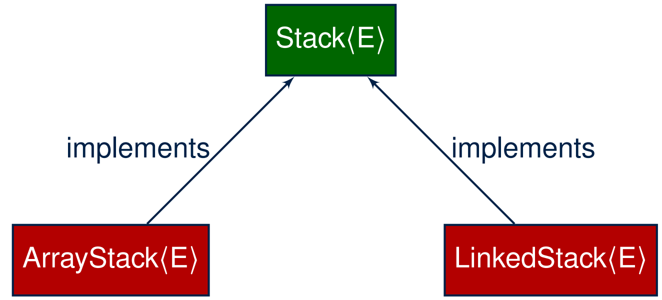
Operation	Return Value	Stack Contents
–	–	∅
isEmpty	*true*	∅
push(5)	–	5
push(3)	–	3 5
push(1)	–	1 3 5
size	3	1 3 5
top	1	1 3 5
pop	1	3 5
pop	3	5
pop	5	∅

Generic Stack: Interface

```
public interface Stack< E > {
  public int size();
  public boolean isEmpty();
  public E top();
  public void push( E e);
  public E pop();
}
```

The ***Stack*** ADT, declared as an ***interface***, allows ***alternative implementations*** to conform to its method headers.

Generic Stack: Architecture



Implementing Stack: Array (1)

```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int t; /* index of top */
  public ArrayStack() {
    data = (E[]) new Object[MAX_CAPACITY];
    t = -1;
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  public E top() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[t]; }
  }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { t ++; data[t] = e; }
  }
  public E pop() {
    E result;
    if (isEmpty()) { /* Precondition Violated */ }
    else { result = data[t]; data[t] = null; t --; }
    return result;
  }
}
```

Implementing Stack: Array (2)

- Running Times of *Array*-Based **Stack** Operations?

***ArrayStack* Method**	**Running Time**
size	O(1)
isEmpty	O(1)
top	O(1)
push	O(1)
pop	O(1)

- **Exercise** This version of implementation treats the ***end*** of array as the ***top*** of stack. Would the RTs of operations change if we treated the ***beginning*** of array as the ***top*** of stack?
- **Q**. What if the preset capacity turns out to be insufficient?

  A. `IllegalArgumentException` occurs and it takes ***O(1)*** time to respond.
- At the end, we will explore the alternative of a ***dynamic array***.

Implementing Stack: Singly-Linked List (1)

```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Question:

Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

Which ***implementation strategy*** should be chosen?

Implementing Stack: Singly-Linked List (2)

- If the ***front of list*** is treated as the ***top of stack***, then:
  - All stack operations remain ***O(1)*** [∵ removeFirst takes ***O(1)***]
- If the ***end of list*** is treated as the ***top of stack***, then:
  - The ***pop*** operation takes ***O(n)*** [∵ removeLast takes ***O(n)***]
- But in both cases, given that a linked, ***dynamic*** structure is used, ***no resizing*** is necessary!

Generic Stack: Testing Implementations

```
@Test
public void testPolymorphicStacks() {
  Stack<String> s = new ArrayStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());

  s = new LinkedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
}
```

Polymorphism & Dynamic Binding

```
1  Stack<String> myStack;
2  myStack = new ArrayStack<String>();
3  myStack.push("Alan");
4  myStack = new LinkedStack<String>();
5  myStack.push("Alan");
```

- *Polymorphism*

  An object may change its ***"shape"*** (i.e., ***dynamic type***) at runtime.

  Which lines? 2, 4
- *Dynamic Binding*

  Effect of a method call depends on the ***"current shape"*** of the target object.

  Which lines? 3, 5

Stack Application: Reversing an Array

- ***Implementing*** a ***generic*** algorithm:

```
public static <E> void reverse(E[] a) {
  Stack<E> buffer = new ArrayStack<E>();
  for (int i = 0; i < a.length; i ++) {
    buffer.push(a[i]);
  }
  for (int i = 0; i < a.length; i ++) {
    a[i] = buffer.pop();
  }
}
```

- ***Testing*** the ***generic*** algorithm:

```
@Test
public void testReverseViaStack() {
  String[] names = {"Alan", "Mark", "Tom"};
  String[] expectedReverseOfNames = {"Tom", "Mark", "Alan"};
  StackUtilities.reverse(names);
  assertArrayEquals(expectedReverseOfNames, names);

  Integer[] numbers = {46, 23, 68};
  Integer[] expectedReverseOfNumbers= {68, 23, 46};
  StackUtilities.reverse(numbers);
  assertArrayEquals(expectedReverseOfNumbers, numbers);
}
```


Stack Application: Matching Delimiters (1)

- **Problem**

  Opening delimiters: `(`, `[`, `{`

  Closing delimiters: `)`, `]`, `}`

  e.g., **Correct**: `()(()){([()])}`

  e.g., **Incorrect**: `({[])}`
- **Sketch of Solution**
  - When a new ***opening*** delimiter is found, ***push*** it to the stack.
  - *Most-recently* found delimiter should be matched first.
  - When a new ***closing*** delimiter is found:
    - If it matches the ***top*** of the stack, then ***pop*** off the stack.
    - Otherwise, an error is found!
  - Finishing reading the input, an empty stack means a success!

Stack Application: Matching Delimiters (2)

- ***Implementing*** the algorithm:

```
public static boolean isMatched(String expression) {
  final String opening = "([{";
  final String closing = ")]}";
  Stack<Character> openings = new LinkedStack<Character>();
  int i = 0;
  boolean foundError = false;
  while (!foundError && i < expression.length()) {
    char c = expression.charAt(i);
    if(opening.indexOf(c) != -1) { openings.push(c); }
    else if (closing.indexOf(c) != -1) {
      if(openings.isEmpty()) { foundError = true; }
      else {
        if (opening.indexOf(openings.top()) == closing.indexOf(c)) { openings.pop(); }
        else { foundError = true; } } }
    i ++; }
  return !foundError && openings.isEmpty(); }
```

- ***Testing*** the algorithm:

```
@Test
public void testMatchingDelimiters() {
  assertTrue(StackUtilities.isMatched(""));
  assertTrue(StackUtilities.isMatched("{[]}({})"));
  assertFalse(StackUtilities.isMatched("{[])"));
  assertFalse(StackUtilities.isMatched("{[]})"));
  assertFalse(StackUtilities.isMatched("({[]}"));
}
```

Stack Application: Postfix Notations (1)

Problem: Given a postfix expression, calculate its value.

Infix Notation	**Postfix** Notation
Operator ***in-between*** Operands	Operator ***follows*** Operands
Parentheses force precedence	Order of evaluation embedded
`3`	`3`
`3 + 4`	`3 4 +`
`3 + 4 + 5`	`3 4 + 5 +`
`3 + (4 + 5)`	`3 4 5 + +`
`3 - 4 * 5`	`3 4 5 * -`
`(3 - 4) * 5`	`3 4 - 5 *`

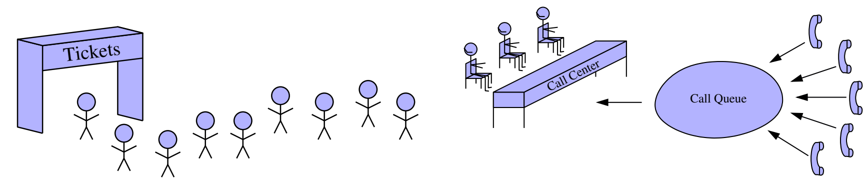
Stack Application: Postfix Notations (2)

Sketch of Solution

- When input is an ***operand*** (i.e., a number), ***push*** it to the stack.
- When input is an ***operator***, obtain its two ***operands*** by ***popping*** off the stack **twice**, evaluate, then ***push*** the result back to stack.
- When finishing reading the input, there should be **only one** number left in the stack.
- **Error** if:
  - Not enough items left in the stack for the operator [e.g., `523+**`]
  - When finished, two or more numbers left in stack [e.g., `53+6`]

What is a Queue?

- A ***queue*** is a collection of objects.
- Objects in a ***queue*** are inserted and removed according to the *first-in, first-out (**FIFO**)* principle.
  - Each new element joins at the ***back***/***end*** of the queue.
  - ***Cannot*** access arbitrary elements of a queue
  - ***Can*** only access or remove the *least-recently inserted (or longest-waiting)* element



The Queue ADT

- ***first*** ≈ ***top*** of stack
  - [***precondition***: queue is not empty]
  - [***postcondition***: return item **first** enqueued]
- ***size***
  - [***precondition***: **none**]
  - [***postcondition***: return number of items enqueued]
- ***isEmpty***
  - [***precondition***: **none**]
  - [***postcondition***: return whether there is no item in the queue]
- ***enqueue(item)*** ≈ ***push*** of stack
  - [***precondition***: queue is not full]
  - [***postcondition***: enqueue item as the "last" of the queue]
- ***dequeue*** ≈ ***pop*** of stack
  - [***precondition***: queue is not empty]
  - [***postcondition***: remove and return the first of the queue]

Queue: Illustration

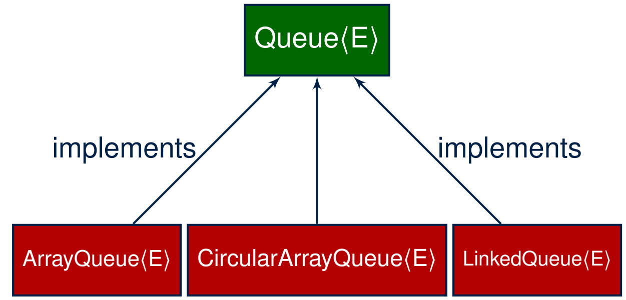
Operation	Return Value	Queue Contents
–	–	∅
isEmpty	*true*	∅
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
enqueue(1)	–	(5, 3, 1)
size	3	(5, 3, 1)
dequeue	5	(3, 1)
dequeue	3	1
dequeue	1	∅

Generic Queue: Interface

```
public interface Queue< E > {
  public int size();
  public boolean isEmpty();
  public E first();
  public void enqueue( E e);
  public E dequeue();
}
```

The ***Queue*** ADT, declared as an ***interface***, allows ***alternative implementations*** to conform to its method headers.

Generic Queue: Architecture



Implementing Queue ADT: Array (1)

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r; /* rear index */
  public ArrayQueue() {
    data = (E[]) new Object[MAX_CAPACITY];
    r = -1;
  }
  public int size() { return (r + 1); }
  public boolean isEmpty() { return (r == -1); }
  public E first() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[0]; }
  }
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { r ++; data[r] = e; }
  }
  public E dequeue() {
    if (isEmpty()) { /* Precondition Violated */ }
    else {
      E result = data[0];
      for (int i = 0; i < r; i ++) { data[i] = data[i + 1]; }
      data[r] = null; r --;
      return result;
    }
  }
}
```

Implementing Queue ADT: Array (2)

- Running Times of *Array*-Based *Queue* Operations?

ArrayQueue **Method**	**Running Time**
size	O(1)
isEmpty	O(1)
first	O(1)
enqueue	O(1)
dequeue	$O(n)$

- **Exercise** This version of implementation treats the ***beginning*** of array as the ***first*** of queue. Would the RTs of operations change if we treated the ***end*** of array as the ***first*** of queue?
- **Q**. What if the preset capacity turns out to be insufficient?

  A. `IllegalArgumentException` occurs and it takes ***O(1)*** time to respond.
- At the end, we will explore the alternative of a ***dynamic array***.

Implementing Queue: Singly-Linked List (1)

```
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Question:

Queue Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first	list.last
enqueue	list.addLast	list.addFirst
dequeue	list.removeFirst	list.removeLast

Which ***implementation strategy*** should be chosen?

Implementing Queue: Singly-Linked List (2)

- If the ***front of list*** is treated as the ***first of queue***, then:
  - All queue operations remain ***O(1)*** [$\because$ removeFirst takes ***O(1)***]
- If the ***end of list*** is treated as the ***first of queue***, then:
  - The ***dequeue*** operation takes ***O(n)*** [$\because$ removeLast takes ***O(n)***]
- But in both cases, given that a linked, ***dynamic*** structure is used, ***no resizing*** is necessary!

Generic Queue: Testing Implementations

```
@Test
public void testPolymorphicQueues() {
  Queue<String> q = new ArrayQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());

  q = new LinkedQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());
}
```

Polymorphism & Dynamic Binding

```
1 Queue<String> myQueue;
2 myQueue = new CircularArrayQueue<String>();
3 myQueue.enqueue("Alan");
4 myQueue = new LinkedQueue<String>();
5 myQueue.enqueue("Alan");
```

- *Polymorphism*

  An object may change its ***"shape"*** (i.e., *dynamic type*) at runtime.

  Which lines? 2, 4
- *Dynamic Binding*

  Effect of a method call depends on the *"current shape"* of the target object.

  Which lines? 3, 5

Exercise: Implementing a Queue using Two Stacks

```
public class StackQueue<E> implements Queue<E> {
  private Stack<E> inStack;
  private Stack<E> outStack;
  ...
}
```

- For `size`, add up sizes of `inStack` and `outStack`.
- For `isEmpty`, are `inStack` and `outStack` both empty?
- For `enqueue`, ***push*** to `inStack`.
- For `dequeue`:
  - ***pop*** from `outStack`

    If `outStack` is empty, we need to first ***pop*** all items from `inStack` and ***push*** them to `outStack`.

Exercise: Why does this work? [***implement*** and ***test***]

Exercise: Running Time? [see analysis on ***dynamic arrays***]

Optional Materials

These topics are useful for your knowledge about ADTs, stacks, and Queues.

You are **encouraged** to follow through these online lectures:

`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2011_W22`

- ***Design by Contract*** and ***Modularity***
  - Week 5: Lecture 3, Parts A2 - A3
- ***Circular Arrays*** and ***Double-Ended Queue***
  - Week 6: Lecture 3, Parts D3 – D5
- ***Dynamic Arrays*** and ***Amortized Analysis***
  - Week 6: Lecture 3, Parts E1 - E5

Terminology: Contract, Client, Supplier

- A *supplier* implements/provides a service (e.g., microwave).
- A *client* uses a service provided by some supplier.
  - The client is required to follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
  - If instructions are followed, the client would **expect** that the service does what is guaranteed (e.g., a lunch box is heated).
  - The client does not care how the supplier implements it.
- What are the ***benefits*** and ***obligations*** of the two parties?

	benefits	***obligations***
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	provide a service

- There is a *contract* between two parties, violated if:
  - The instructions are not followed. [Client's fault]
  - Instructions followed, but service not satisfactory. [Supplier's fault]

Client, Supplier, Contract in OOP (1)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
} }
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();
    m.heat(obj);
} }
```

Method call ***m.heat(obj)*** indicates a client-supplier relation.

- **Client**: resident class of the method call [`MicrowaveUser`]
- **Supplier**: type of context object (or call target) **m** [`Microwave`]

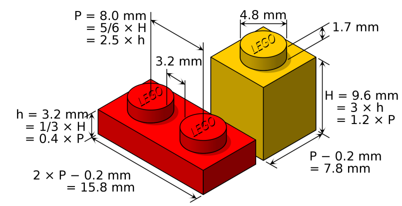
Client, Supplier, Contract in OOP (2)

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */}}
```

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();
    m.heat(obj);
} }
```

- The *contract* is *honoured* if:
  - Right **before** the method call:
    - State of `m` is as assumed: `m.on==true` and `m.locked==ture`
    - The input argument `obj` is valid (i.e., not explosive).
  - Right **after** the method call: `obj` is properly heated.
- If any of these fails, there is a *contract violation*.
  - `m.on` or `m.locked` is `false` ⇒ `MicrowaveUser`'s fault.
  - `obj` is an explosive ⇒ `MicrowaveUser`'s fault.
    A fault from the client is identified ⇒ Method call will not start.
  - Method executed but `obj` not properly heated ⇒ `Microwave`'s fault

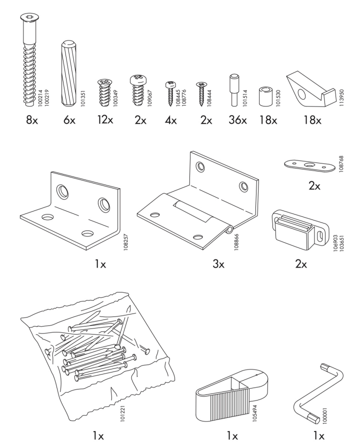
Modularity (1): Childhood Activity

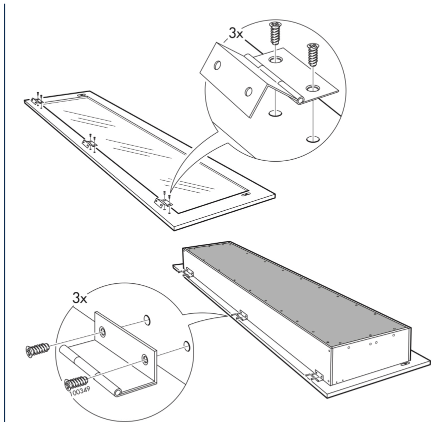


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

Sources: https://commons.wikimedia.org and https://www.wish.com

Modularity (2): Daily Construction



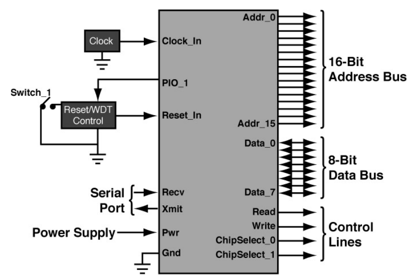


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

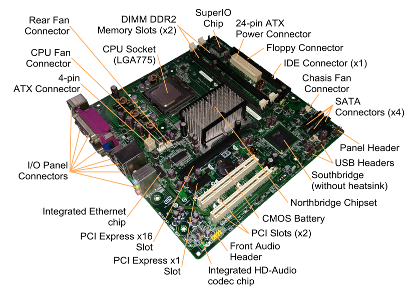
Source: https://usermanual.wiki/

Modularity (3): Computer Architecture

Motherboards are built from functioning units (e.g., *CPUs*).



(INTERFACE) SPECIFICATION



(ASSEMBLY) ARCHITECTURE

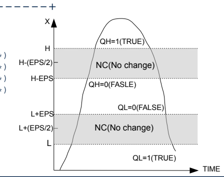
Sources: www.embeddedlinux.org.cn and https://en.wikipedia.org

Modularity (4): System Development

Safety-critical systems (e.g., *nuclear shutdown systems*) are built from *function blocks*.

```
(* DECLARATION *)
          +----------+
          | LIMITS_  |
          | ALARM    |
    REAL--|H       QH|--BOOL
    REAL--|X        Q|--BOOL
    REAL--|L       QL|--BOOL
    REAL--|EPS       |
          +----------+
```

```
FUNCTION_BLOCK LIMITS_ALARM
  VAR_INPUT
    H   : REAL; (* High limit     *)
    X   : REAL; (* Variable value *)
    L   : REAL; (* Lower limit    *)
    EPS : REAL; (* Hysteresis     *)
  END_VAR
  VAR_OUTPUT
    QH : BOOL; (* High flag    *)
    Q  : BOOL; (* Alarm output *)
    QL : BOOL; (* Low flag     *)
  END_VAR
END_FUNCTION_BLOCK
```



(INTERFACE) SPECIFICATION

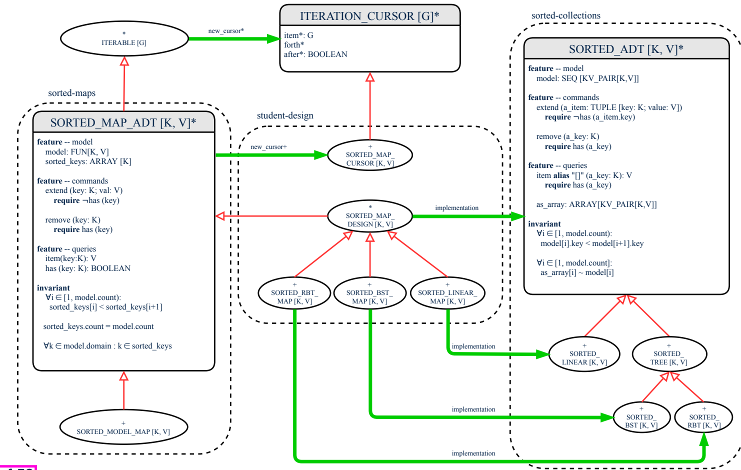
```
(* Function block body in FBD language *)
                        HIGH_ALARM
                    +-------------+
                    | HYSTERESIS  |
X-----------------+--|XIN1       Q|--+----------QH
          +----+ w2|  |            |  |
H---------|  - |------|XIN2        |  |
      +---|    |   |  |            |  |
      |   +----+   |  |            |  |
      +---------------|EPS         |  |   +------+
  +----+w1|        |  +-------------+  +--| >=1  |
EPS --| / |--|     |                      |      |--Q
2.0 --|   |  |     |     LOW_ALARM     +--|      |
  +----+  |        |  +-------------+  |  +------+
      |   +----+ w3|  | HYSTERESIS  |  |
L ---------|  + |------|XIN1       Q|--+-----------QL
      |   |    |   |  |             |
      +---|    |   +--|XIN2         |
      |   +----+      |             |
      +---------------|EPS          |
                      +-------------+
```

(ASSEMBLY) ARCHITECTURE

Sources: https://plcopen.org/iec-61131-3

Modularity (5): Software Design

Software systems are composed of ***well-specified classes***.



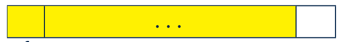
Design Principle: Modularity

- *Modularity* refers to a sound quality of your design:
  1. **Divide** a given complex ***problem*** into inter-related ***sub-problems*** via a logical/justifiable functional decomposition.
     e.g., In designing a game, solve sub-problems of: 1) rules of the game; 2) actor characterizations; and 3) presentation.
  2. **Specify** each ***sub-solution*** as a *module* with a clear **interface**: inputs, outputs, and **input-output relations**.
     - The UNIX principle: Each command does one thing and does it well.
     - In objected-oriented design (OOD), each class serves as a module.
  3. **Conquer** original ***problem*** by assembling ***sub-solutions***.
     - In OOD, classes are assembled via client-supplier relations (aggregations or compositions) or inheritance relations.
- A *modular design* satisfies the criterion of modularity and is:
  - ***Maintainable***: fix issues by changing the relevant modules only.
  - ***Extensible***: introduce new functionalities by adding new modules.
  - ***Reusable***: a module may be used in different compositions
- Opposite of modularity: A *superman module* doing everything.

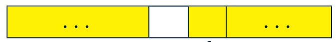
Implementing Queue ADT: Circular Array (1)

- Maintain two indices: ***f*** for ***front***; ***r*** for ***next available slot***.
- **Maximum size**: $N - 1$ [N = `data.length`]
- **Empty Queue**: when ***r = f***

- **Full Queue**: when ***((r + 1) % N) = f***
  - When $r > f$:



  - When $r < f$:



- **Size of Queue**:
  - If $r = f$: ***0***
  - If $r > f$: ***r - f***

  - If $r < f$: ***r + (N - f)***

Implementing Queue ADT: Circular Array (2)

Running Times of *CircularArray*-Based *Queue* Operations?

***CircularArrayQueue* Method**	**Running Time**
size	O(1)
isEmpty	O(1)
first	O(1)
enqueue	O(1)
dequeue	*O*(1)

Exercise: Create a Java class `CircularArrayQueue` that `implements` the `Queue` interface using a ***circular array***.

Limitations of Queue

- Say we use a ***queue*** to implement a ***waiting list***.
  - What if we `dequeue` the front customer, but find that we need to ***put them back to the front*** (e.g., seat is still not available, the table assigned is not satisfactory, *etc.*)?
  - What if the customer at the end of the queue decides not to wait and leave, how do we ***remove them from the end of the queue***?
- **Solution:** A new ADT extending the ***Queue*** by supporting:
  - ***insertion*** to the ***front***
  - ***deletion*** from the ***end***

The Double-Ended Queue ADT

- ***Double-Ended Queue*** (or ***Deque***) is a queue-like data structure that supports ***insertion*** and ***deletion*** at both the ***front*** and the ***end*** of the queue.

```
public interface Deque<E> {
  /* Queue operations */
  public int size();
  public boolean isEmpty();
  public E first();
  public void addLast(E e); /* enqueue */
  public E removeFirst(); /* dequeue */
  /* Extended operations */
  public void addFirst(E e);
  public E removeLast();
}
```

- **Exercise**: Implement *Deque* using a ***circular array***.
- **Exercise**: Implement *Deque* using a ***SLL*** and/or ***DLL***.

Array Implementations: Stack and Queue

- When implementing ***stack*** and ***queue*** via ***arrays***, we imposed a maximum capacity:

```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ... }
  }
  ...
}
```

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ...
  }
  ...
}
```

- This made the ***push*** and ***enqueue*** operations both cost ***O(1)***.

Dynamic Array: Constant Increments

Implement ***stack*** using a ***dynamic array*** resizing itself by a constant increment:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int C;
4    private int capacity;
5    private E[] data;
6    public ArrayStack() {
7      I = 1000; /* arbitrary initial size */
8      C = 500; /* arbitrary fixed increment */
9      capacity = I;
10     data = (E[]) new Object[capacity];
11     t = -1;
12   }
13   public void push(E e) {
14     if (size() == capacity) {
15       /* resizing by a fixed constant */
16       E[] temp = (E[]) new Object[capacity + C];
17       for(int i = 0; i < capacity; i ++) {
18         temp[i] = data[i];
19       }
20       data = temp;
21       capacity = capacity + C
22     }
23     t++;
24     data[t] = e;
25   }
26 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by a ***constant*** amount.
- **L17** – **L19** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L14 – L22** as the ***resizing*** part and **L23 – L24** as the ***update*** part.

Dynamic Array: Doubling

Implement ***stack*** using a ***dynamic array*** resizing itself by doubling:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int capacity;
4    private E[] data;
5    public ArrayStack() {
6      I = 1000; /* arbitrary initial size */
7      capacity = I;
8      data = (E[]) new Object[capacity];
9      t = -1;
10   }
11   public void push(E e) {
12     if (size() == capacity) {
13       /* resizing by doubling */
14       E[] temp = (E[]) new Object[capacity * 2];
15       for(int i = 0; i < capacity; i ++) {
16         temp[i] = data[i];
17       }
18       data = temp;
19       capacity = capacity * 2
20     }
21     t++;
22     data[t] = e;
23   }
24 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by ***doubling*** its current size.
- **L15** – **L17** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L12 – L20** as the resizing part and **L21 – L22** as the update part.

Avg. RT: Const. Increment vs. Doubling

- Without loss of generality, assume: There are n ***push*** operations, and the **last push** triggers the **last** ***resizing*** routine.

	Constant Increments	Doubling
RT of exec. update part for n pushes	$O(n)$	
RT of executing 1st resizing	I	
RT of executing 2nd resizing	$I + C$	$2 \cdot I$
RT of executing 3rd resizing	$I + 2 \cdot C$	$4 \cdot I$
RT of executing 4th resizing	$I + 3 \cdot C$	$8 \cdot I$
RT of executing k^{th} resizing	$I + (k-1) \cdot C$	$2^{k-1} \cdot I$
RT of executing last resizing	n	
# of resizing needed (solve k for $RT = n$)	$O(n)$	$O(log_2 n)$
Total RT for n pushes	$O(n^2)$	$O(n)$
Amortized/Average RT over n pushes	$O(n)$	$O(1)$

- Over n push operations, the *amortized* / *average* running time of the ***doubling*** strategy is more efficient.

Beyond this lecture . . .

- Attempt the exercises throughout the lecture.
- Implement the ***Postfix Calculator*** using a stack.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)