Interfaces, Iterators, Recursion
Dynamic Data Structures

Reading Assignment
Chapters 4-5

Interfaces

- Interfaces contain
  - `abstract` methods
  - `public static final` fields (i.e., constants)
- All members of an `interface` are `public`
- A Java interface can be used in the same way as a Java class
- Both interface and class names are types in Java
- Just like abstract classes, interfaces cannot be instantiated
- Interfaces are basically abstract classes except
  - All methods in an `interface` are `abstract`
  - An `abstract` class may contain non-abstract methods
  - Thus, some methods of an `abstract class` may be implemented
  - No methods of an `interface` is implemented
- A class `implements` an `interface` by
  - Declaring that it `implements` the `interface`
  - Defining implementations of all the `interface` methods

Interfaces

- A class is `implements` an interface if it implements all abstract methods specified by an interface
  `class X implements I { … }`
- An interface may `extend` one or more interfaces
  `interface Stack extends List, Comparable { … }`
  `interface Container extends Collection { … }`
- When a class `implements` an interface method, it must implement its exact signature
- Interfaces form their own inheritance hierarchy
- A class may implement multiple interfaces
  - This is essentially multiple inheritance
    `class X implements I1, I2 { … }`
    `class X extends A implements I1, I2 { … }`
- The relationships induced by `extends` and `implements` are all is-a relationships

Defining and implementing interface Comparable

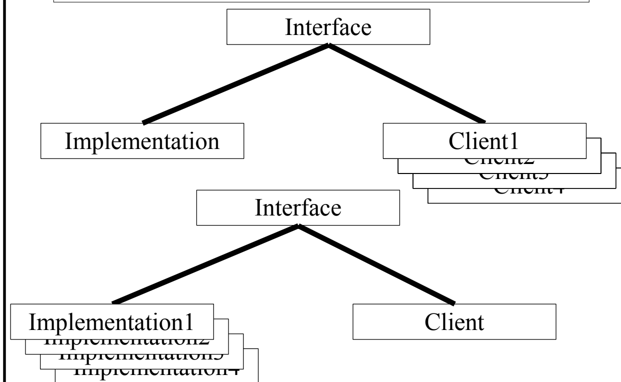
```
public interface Comparable {
  int compareTo(Object x);
}

public abstract class Shape implements Comparable {
  public abstract double area();
  public int compareTo(Object x) {
    Shape s = (Shape)x;
    double diff = area() - s.area();
    if (diff == 0) return 0;
    else if (diff < 0) return -1;
    else return 1;
  }
}
```

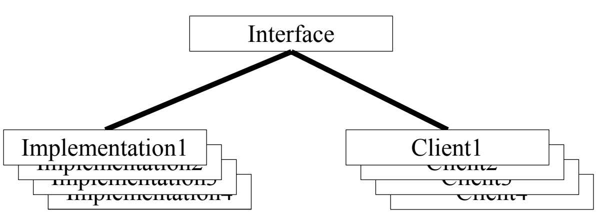
Separating the What from the How

- *Complexity* is a big problem in software engineering
- We can control complexity by:
  - Separating concerns
  - Breaking software into smaller, simpler pieces
  - Making sure that each piece knows only *what* other pieces do, not *how* they do it
    - Abstraction, information hiding, encapsulation
    - High coupling within components
    - Low coupling among components
- Benefits of this approach:
  - ease of use
  - ease of modification
  - Ease of maintenance and evolution
- Java interfaces separate the *what* from the *how* (*reduce coupling*)
- Java classes *encapsulate* all that is necessary to implement an interface (*increase cohesion*)

Interface relationships



Interface relationships



- Sorting interface
  - Implemented by different sorting algorithms (e.g., Quicksort, Heapsort, Insertionsort, Bubblesort, Mergesort, Radixsort)
  - Used by different clients to sort Strings, integers, doubles, dates, records

Extending and contracting interfaces

- A given interface is often extended or contracted
- Extension
  - Operations are added to an interface
  - A standard interface is fine, but some additional operations might be useful
  - Inheritance relationships (e.g., Employee, Shape hierarchies)
- Contraction
  - Operations are taken away to make the interface simpler
  - This is also known as the Adapter design pattern
  - Ex: a Stack or Queue interface is simpler than a general list interface

- Seminal paper on the subject
  - D. L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transaction on Software Engineering,* Vol. SE-5, No 2, 1979.

Genericity

- An important goal of object-oriented programming is code reuse
- An important mechanism is genericity
- If two algorithms are identical except for the basic type, then a generic implementation can be used to describe the functionality
- For example
  - A swap routine is the same regardless of the type of the elements being swaped
  - A sorting algorithm is independent of the element type provided there is a routine to compare two elements of this type
- Generic mechanisms in programming languages
  - C++ provides templates to implement generics of this sort
  - Ada provides context-sensitive macros
  - C provides macros

Genericity mechanisms in Java

- Java provides several mechanisms for different kinds of generics
  - Object, the superclass of all classes
    - Ex: generic swap routine
  - Interface types
    - Ex: search or sort object of type **`interface Comparable`**
  - Function objects, functors, or function pointers
    - Ex: generic sort routine with Comparator object

A bad swap() routine

- Defining a generic swap routine does **not** work in Java since references are passed by value.

```
void swap(Object a, Object b) {
  Object temp = a; a = b; b = temp;
}
```

- Using a generic swap routine

```
String s1 = new String("ABC");
String s2 = new String("XYZ");
swap(s1, s2);
// s1 and s2 still refer to ABC and XYZ, respectively

// use in-line swap instead
String temp = s1; s1 = s2; s2 = temp; // this works ☺
```


The class Object

- Object is the root class of all Java classes
- Thus, Object is a superclass for every Java class
- Object is not an abstract call, that is, it provides implementations for all of its methods

- Here is a subset of the methods defined in **`class Object`**

```
public class Object {
  public Object() { … }
  public String toString() { … }
  public boolean equals(Object x) { … }
  public int hashCode() { … }
  protected Object clone() { … }
}
```

Wrappers for primitive types

- Primitive types (i.e., `byte, short, int, long, float, double, char, boolean`) are not compatible with the reference type `Object`
- Thus, values of primitive cannot be passed to parameters of type `Object`
- To get around this problem, Java provides wrapper classes for all primitive types (i.e., `Byte, Short, Integer, Long, Float, Double, Character, Boolean)`

```
public final class Integer implements Comparable {
  private int value;
  public Integer(int x) { value = x; }
  public int intValue() { return value; }
  public String toString() { return "" + value; }
  public int compareTo() { … }
}
Integer k1 = new Integer(17);
Integer k2 = new Integer(37);
swap(k1, k2);
```


Homogeneous and heterogeneous collections

- Almost all collections are defined to contain objects
  - As a result, any object can be put into a collection
  - Use the `instanceof` operator to determine the kind of object during retrieval (see list main program below)
    `x instanceof String`
  - Cast the instance to the correct type accordingly
    `String s = (String)x;`
- Collections are heterogeneous or homogeneous
  - Homogeneous: all components are of the same type
  - Heterogeneous: components may be of different types
- Most collections in Java are heterogeneous
  - Enforcing homogeneity can be done easily at run-time
  - Enforcing homogeneity at compile-time is hard in Java

Interface genericity: linear search

```
public class InterfaceGenericityLinearSearch {
  public static Comparable findMax(Comparable[] a) {
    // a is an array of Comparables
    // for objects that implement the Comparable interface
    // and thus implement the method compareTo()
    int maxIndex = 0;
    for (int k = 1; k < a.length; k++) {
      if (a[k].compareTo(a[maxIndex]) > 0) maxIndex = k;
    }
    return a[maxIndex];
  }
  public static void main(String[] args) {
    Shape[] sha = { new Rectangle(10, 20), Circle(33) };
    String[] str = { "Bette", "Hausi", "Peggy", "Maarten" };
    System.out.println(findMax(sha));
    System.out.println(findMax(str));
  }
}
```


Function object genericity: linear search

```
public interface Comparator {
    public int compare(Object lhs, Object rhs);
}
public class OrderByLastStringLetter implements Comparator {
  public int compare(Object lhs, Object rhs) {
    String s1 = (String)lhs; String s2 = (String)rhs;
    return s1.charAt(s1.length()-1) - s2.charAt(s2.length()-1);
  }
}
public class FunctionObjectGenericityLinearSearch {
  public static Object findMax(Object[] a, Comparator cp) {
    int maxIndex = 0;
    for (int k = 1; k < a.length; k++)
      if (cp.compare(a[maxIndex], a[k]) > 0) maxIndex = k;
    return a[maxIndex];
  }
  public static void main(String[] args) {
    Object[] strs = new Object[3];
    strs[0] = new String("Bette");    strs[1] = new String("Hausi");
    strs[2] = new String("Peggy");    strs[3] = new String("Maarten");
    System.out.println(findMax(strs), new OrderByLastStringLetter()));
  }
}
```

Recursive algorithms and data structures

- A method (algorithm) or class (data structure) that is partially defined in terms of itself is called *recursive*
- Recursion is a powerful algorithm design and programming tool that can lead to elegant and efficient algorithms and data structures
- Recursion is based on the principle of mathematical induction
- Thus, a recursive algorithm consists of
  - a base case
  - an induction step

Recursive methods and classes

- A recursive method is a method that directly or indirectly calls itself
- Shortest and simplest direct and indirect recursive methods; note that both examples result in infinite recursion since there is no base case

```
void a() { a(); } // direct recursion
void a() { b(); } void b() { a(); } // indirect
```

- Shortest and simplest direct and indirect recursive classes

```
class X { X x; } // direct recursion
class X { Y y; } class Y { X x; } // indirect
```


Compute the sum of k integers recursively and iteratively

```
void recursiveAlgo(int n) {
  if ("simplest case") {
    // base case
    "solve directly"
    "for example for n=1"
  } else {
    // induction step
    "make a recursive call
     with simpler case"
    "for example for n-1"
  }
}
```

- **Recursive algorithm**

```
int sum(int k) {
  if (k==1) return 1;
  else return sum(k-1) + k;
}
```

- **Template for a recursive method**
- **Base case is a simple case where we know the solution**
- **For the induction step, we assume that we know the solution for a previous solution, say n-1, and compute the solution in terms of this solution**
- **For example, if we know the sum of the first n-1 integers (i.e., sum(n-1)), the sum of n integers is n + sum(n-1)**

- **Iterative algorithm**

```
int sum(int k) {
  int s = 0;
  for (int j=1; j<=k; j++)
    s = s + j;
  return s;
}
```


Run-time stack

- Every recursive algorithm can be converted into a non-recursive or iterative algorithm by simulating the run-time stack
- The run-time stack consists of *activation records* or *stack frames*
- An activation record contains the following information
  - Return address (address of caller)
  - Destination address (address of callee)
  - Actual parameters (parameters being passed)
  - Local variables (local variables of the routine being called)
- Whenever a method call is made, a new activation record is allocated and *pushed* onto the run-time stack
- When a call returns, its record is *popped* off the run-time stack

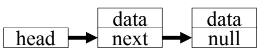
Recursive data structures

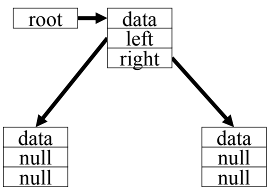
```
public class ListNode {
  private Object data;
  private Node next;

  "method definitions"
}
```

```
public class TreeNode {
  private Object data;
  private Node left;
  private Node right;

  "method definitions"
}
```





A generic Node class

```
public class Node {
  private Object data;
  private Node next;
  private Node prev;

public Node(Object data,
  Node next, Node prev) {
  this.data = data;
  this.next = next;
  this.prev = prev;
}
public Node() {
  this(null, null, null);
}
public Node(Object data) {
  this(data, null, null);
}
```

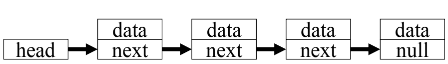
```
public Object getData() {
  return data;
}
public Node getNext() {
  return next;
}
public Node getPrev() {
  return prev;
}
public void setData(Object data) {
  this.data = data;
}
public void setNext(Node next) {
  this.next = next;
}
public void setPrev(Node prev) {
  this.prev = prev;
}
}
```


Singly linked list

```
public class SLinkedList {
  private Node head;
  private int size;

  public SLinkedList() {
    head = null;
    size = 0;
  }
  public Node getHead() {
    return head;
  }
  public boolean isEmpty() {
    return head == null;
  }
```

```
public int size() {
    return size;
  }
public Object getFirst() {
  if (head==null) {
    return null;
  } else {
    return head.getData();
  }
}
```



Singly linked list

```
public void
  insertFirst(Object data) {
  Node node = new Node(data);
  size++;
  if (head == null) {
    head = node; tail = node;
  } else {
    node.setNext(head);
    head = node;
  }
}
```

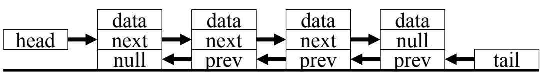
```
public Object deleteFirst() {
  if (head==null){
    return null;
  } else {
    Node aux = head;
    head = aux.getNext();
    aux.setNext(null);
    if (head==null) {
      tail = null;
    } else {
      head.setPrev(null);
    }
    size--;
    return aux.getData();
  }
}
```

Doubly linked list class

```
public class LinkedList {
  private Node head;
  private Node tail;
  private int size;

  public LinkedList() {
    head = null; tail = null;
    size = 0;
  }
  public Node getHead() {
    return head; }
  public Node getTail() {
    return tail; }
  public boolean isEmpty() {
    return head == null; }
```

```
public int size() {
  return size;
}
public Object getFirst() {
  if (head==null) {
    return null;
  } else {
    return head.getData();
  }
}
public Object getLast() {
  if (tail==null) {
    return null;
  } else {
    return tail.getData();
  }
}
```



Methods InsertFirst and InsertLast

```
public void
  insertFirst(Object data) {
  Node node = new Node(data);
  size++;
  if (head == null) {
    head = node; tail = node;
  } else {
    node.setNext(head);
    head = node;
  }
}
```

```
public void
  insertLast(Object data) {
  Node node = new Node(data);
  size++;
  if (tail == null) {
    head = node; tail = node;
  } else {
    node.setNext(tail);
    tail = node;
  }
}
```

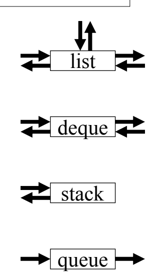

Methods DeleteFirst and DeleteLast

```
public Object
  deleteFirst() {
  if (head==null){
    return null;
  } else {
    Node aux = head;
    head = aux.getNext();
    aux.setNext(null);
    if (head==null) {
      tail = null;
    } else {
      head.setPrev(null);
    }
    size--;
    return aux.getData();
  }
}
```

```
public Object
  deleteLast() {
  if (tail==null) {
    return B;
  } else {
    Node aux = tail;
    tail = aux.getPrev();
    aux.setPrev(null);
    if (tail==null) {
      head = null;
    } else {
      tail.setNext(null);
    }
    size--;
    return aux.getData();
  }
}
```


Lists, stacks, queues, and deques

- List
  - insertFirst, insertLast, deleteFirst, deleteLast, isEmpty
- Deque (double-ended queues)
  - insertFirst, insertLast, deleteFirst, deleteLast, isEmpty
- Stack
  - push, pop, top, empty
  - Inser and delete at the same end
- Queue
  - enqueue, dequeue, front, empty
  - Insert at one end, delete at the other end



Stack and queue interfaces

```
public interface Stack {
  void push(Object data);
  Object pop();
  Object top();
  boolean isEmpty();
  int size();
}
```

```
public interface Queue {
  void enqueue(Object data);
  Object dequeue();
  Object front();
  boolean isEmpty();
  int size();
}
```

- Stack applications
- Stack of plates in cafeteria
- Run-time stack
- Recursion
- Evaluating expressions
- Balanced parentheses
- Postfix notation
- LIFO (Last-in, first-out)

- Queue applications
- Check out line at store
- Car wash
- Network queues
- Pipes and filters
- Traffic simulation
- FIFO (First-in, First-out)

Stack and queue definitions

```
public class LinkedStack
  extends LinkedList
  implements Stack {

  public LinkedStack() {
    super();
  }
  public Object pop() {
    return deleteFirst();
  }
  public void push(Object data) {
    insertFirst(data);
  }
  public Object top() {
    return getFirst();
  }
  // methods size() and isEmpty()
  // are inherited from LinkedList
}
```

```
public class LinkedQueue
  extends LinkedList
  implements Queue {

  public LinkedQueue() {
    super();
  }
  public Object dequeue() {
    return deleteLast();
  }
  public void enqueue(Object data) {
    insertFirst(data);
  }
  public Object front() {
    return getLast();
  }
  // methods size() and isEmpty()
  // are inherited from LinkedList
}
```


Prefix, infix and postfix notation

- Prefix notation
  - Method, function, procedure calls
  - abc(17, 34), area(22), area(xyz(6))
- Infix notation
  - Arithmetic expression
  - 3 + 15, a + b*c
- Postfix notation
  - HP calculators
  - Reverse Polish notation
  - K++

Prefix	Infix	Postfix	Result
34	34	34	34
+ 34 22	34 + 22	34 22 +	56
* 22 2 + 34	34 + 22 * 2	34 22 2 * +	78
* 34 22 + 2	34 * 22 + 2	2 34 22 * +	750
+ 34 22 * 2	(34 + 22) * 2	34 22 + 2 *	112

Test program for class LinkedStack

```
public static void main(String[] args) {
   Stack s = new LinkedStack();
   s.push(new Integer(4)); s.push(new Integer(7));
   s.push(new Integer(9)); s.push(new String("CSc115"));
   s.push(new String("CSc160")); s.push(new Double(3.14));

   while (!s.isEmpty()) {
      Object obj = s.pop();
      System.out.print("Size = " + s.size() + " Obj = ");
      if (obj instanceof Integer) {
         int k = ((Integer)obj).intValue();
         System.out.println(k);
      } else if (obj instanceof String) {
         String str = (String)obj;
         System.out.println(str);
      } else {
         System.out.println("unknown");
      }
   }
   if (s.isEmpty()) System.out.println("Empty Stack");
}
```

Collections

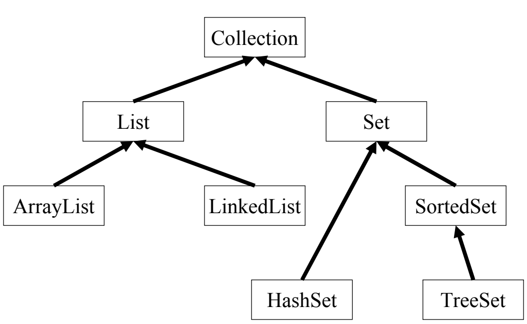
- Collections, data structures, abstract data types (ADTs) consist of two parts
  - data representation
  - operations on those data
- Java provides an entire set of collection APIs
  - interfaces and implementations for fundamental data structures such as lists, stack, queues, deques, trees, graphs
- A container or dictionary is a special collection which supports the operations member , insert, delete, isEmpty
- Here is a simple container interface

```
public interface Container {
   Object member(Object x);
   void insert(Object x);
   Object delete(Object x);
   boolean isEmpty();
}
```


The Java Collection interface

```
public interface Collection {
  boolean add(Object x);
  boolean addAll(Collection c);
  void clear();
  boolean contains(Object x);
  boolean containsAll(Collection c);
  boolean equals(Object x);
  int hashCode();
  boolean isEmpty();
  Iterator iterator();
  boolean remove(Object x);
  boolean removeAll(Collection c);
  boolean retainAll(Collection c);
  int size();
  Object[] toArray();
  Object[] toArray(Object[] a);
}
```


Java Collection hierarchy



Iterator pattern

- An iterator is an object that allows us to enumerate or go through all the elements of a collection or a data structure
- An iterator object controls iteration of the elements of a collection
- The Java collections API uses the iterator pattern extensively
- What if the collection is modified while an iterator is in use?
  - copy: iterator operates on a snapshot taken at instantiation
  - fail-fast: iterator throws an exception on the next method call
  - flexible: iterator tries to adjust to the changes in a manner that varies with the collection type

Iterator and Enumeration interfaces

```
public interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

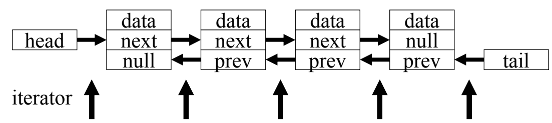
```
public interface Iterator {
  boolean hasNext();
  Object next();
  void remove();
  // removes the last
  // element returned
}
```

```
Vector v = new Vector();
// add objects to v
v.add("abc");
Enumeration itr = v.iterator();
while (itr.hasMoreElements() {
  System.out.
    println(itr.nextElement());
}
```

```
Vector v = new Vector();
// add objects to v
v.add("abc");
Iterator itr = v.iterator();
while (itr.hasNext() {
  System.out.println(itr.next());
}
```


Implementation of iterators

- An iterator object holds is a pointer into the data structure
- The pointer is always between elements, that is, between the element returned last and the element to be returned next
- Thus, for a list the iterator pointer is initialized with **head** or **tail** when the iterator is constructed and advanced with the **next()** or **nextElement()** call
- Since the iterator is defined as a class, multiple iterators can be instantiated from it and hence multiple iterators can be used concurrently on the same data structure



Implementation of iterators

- Iterators can be defined separate from the data structure class or as an inner class of the data structure class
- If an iterator is defined as a separate class, then except for naming conventions and passing the data structure to the constructor as a parameter, it is not clear to which data structure it belongs. Also too much information of the implementation is exposed.
- A better strategy is to define the iterator as an inner class of the data structure. As a result, the iterator is intimately tied to the data structure and the implementation details are nicely hidden (i.e., information hiding software engineering principle is followed).
- We first present the better solution, the inner class solution and then the separate class solution.

Defining an inner class iterator

```
public class LinkedList {
  private class LocalIterator implements Enumeration {
    private Node curNode;
    public LocalIterator() {
      curNode = head;
    }
    public boolean hasMoreElements() {
      return curNode != null;
    }
    public Object nextElement() {
      Object c = curNode;
      curNode = curNode.getNext();
      return ((Node)c).getData();
    }
  }
  public Enumeration iterator() {
    return new LocalIterator();
  }
  // other fields and methods of class LinkedList
}
```

Using the inner class iterator

```
Stack s = new LinkedStack();
s.push(new Integer(4)); s.push(new Integer(7));
s.push(new Integer(9)); s.push(new String("CSc115"));
s.push(new String("CSc160")); s.push(new Double(3.14));
Enumeration litr = ((LinkedList)s).iterator();
while (litr.hasMoreElements()) {
  Object obj = litr.nextElement();
  if (obj instanceof Integer) {
    int k = ((Integer)obj).intValue();
    System.out.println("litr " + k);
  } else if (obj instanceof String) {
    String str = (String)obj;
    System.out.println("litr " + str);
  } else {
    System.out.println("litr unknown");
  }
}
```


Defining a separate class iterator

```
public class LinkedListFwIterator {
  private LinkedList ll;
  private Node curNode;

  public LinkedListFwIterator(LinkedList ll) {
    this.ll = ll;
    this.curNode = ll.getHead();
  }
  public boolean hasNext() {
    return curNode != null;
  }
  public Object next() {
    Object c = curNode;
    curNode = curNode.getNext();
    return ((Node)c).getData();
  }
  public void remove() {
    ll.delete(curNode);
  }
}
```


Using the separate class iterator

```
LinkedListFwIterator gitr =
  new LinkedListFwIterator((LinkedList)s);
while (gitr.hasNext()) {
  Object obj = gitr.next();
  if (obj instanceof Integer) {
    int k = ((Integer)obj).intValue();
    System.out.println("gitr " + k);
  } else if (obj instanceof String) {
    String str = (String)obj;
    System.out.println("gitr " + str);
  } else {
    System.out.println("gitr unknown");
  }
}
```


Summary

- Interfaces
  - Separation of concerns
  - Separating implementation(s) from client(s)
  - Good software engineering
- Recursion
  - Recursive definitions
  - Algorithms (methods) and data structures (classes)
- Dynamic data structures
  - Collections and containers (dictionaries)
  - Genericity, class Object and primitive type wrappers
  - Homogeneous vs. heterogeneous collections
  - Lists, stacks, queues, deques, trees, graphs
- Iterators
  - Iterate over all elements of a data structure
  - Java provides the Enumeration and Iterator interfaces
  - Inner class iterator definition is the best solution