



DM537

Object-Oriented Programming

Peter Schneider-Kamp

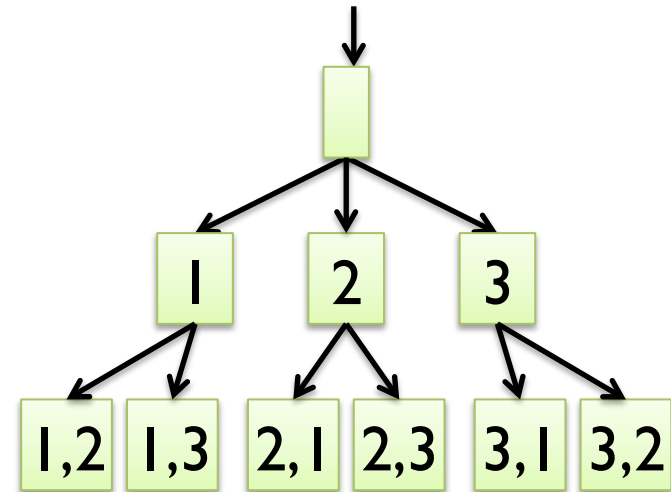
petersk@imada.sdu.dk

<http://imada.sdu.dk/~petersk/DM537/>

MULTIVARIATE TREES

Multivariate Trees

- general class of trees
- nodes can have any number of children
- our application:
 - k-permutations of n
 - all sequences without repetition
 - total of n elements
 - sequences of length $k < n$
- Example:
 - total of $n = 3$ elements
 - sequences of length $k = 2$



MTree ADT: Specification

- data are sequences of integers (`List<Integer>`)
- operations are defined by the following interface

```
public interface MTreeADT extends javax.swing.tree.TreeModel {  
    public Object getRoot();  
    public boolean isLeaf(Object node);  
    public int getChildCount(Object node);  
    public Object getChild(Object parent, int index);  
    public int getIndexOfChild(Object parent, Object child);  
}
```

- `TreeModel` specifies additional operations needed for `JTree`

MTree ADT: Design & Implement.

- Design: use recursive data structure
- Implementation:

```
public class MTreeNode {  
    private List<Integer> seq; // sequence of integers  
    private List<MTreeNode> children =  
        new ArrayList<MTreeNode>();  
    public MTreeNode(List<Integer> seq) { this.seq = seq; }  
    public List<Integer> getSeq() { return this.seq; }  
    public List<MTreeNode> getChildren() { return this.children; }  
    public String toString() { return this.seq.toString(); }  
}
```

MTree ADT: Implementation

- Implementation (continued):

```
public class MTree implements MTreeADT {  
    private MTreeNode root = new MTreeNode(  
        new ArrayList<Integer>());  
    public Object getRoot() { return this.root; }  
    public boolean isLeaf(Object node) {  
        return this.getChildCount(node) == 0;  
    }  
    public int getChildCount(Object node) {  
        return ((MTreeNode)node).getChildren().size();  
    }  
    ...  
}
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...
    public Object getChild(Object parent, int index) {
        return ((MTreeNode)parent).getChildren().get(index);
    }
    public int getIndexOfChild(Object parent, Object child) {
        return ((MTreeNode)parent).getChildren().indexOf(child);
    }
    public void addTreeModelListener(TreeModelListener l) {}
    public void removeTreeModelListener(TreeModelListener l) {}
    public void valueForPathChanged(TreePath p, Object o) {}
    ...
}
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...
    private static MTree makeTree(String title) {
        MTree tree = new MTree();
        JScrollPane content = new JScrollPane(new JTree(tree));
        JFrame window = new JFrame(title);
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setSize(400,400);
        window.setVisible(true);
        return tree;
    } ...
}
```


MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT { ...
    public static void main(String[] args) {
        int n = 4;
        int k = 3;
        MTree tree1 = makeTree("MTree queue");
        tree1.expandQueue(n,k);
        MTree tree2 = makeTree("MTree stack");
        tree2.expandStack(n,k);
        MTree tree3 = makeTree("MTree recursive");
        tree3.expandRecursively(n,k);
    } ...
}
```

MTree ADT: Implementation

- Implementation (continued)

```
public class MTree implements MTreeADT {
```

```
...
```

```
private void expandRecursively(int n, int k) {
```

```
    expandRecursively(this.root, n, k);
```

```
}
```

```
private List<Integer> copyAdd(List<Integer> seq, int i) {
```

```
    List<Integer> copySeq = new ArrayList<Integer>(seq);
```

```
    copySeq.add(i);
```

```
    return copySeq;
```

```
}
```

```
...
```

MTree ADT: Implementation

- Implementation (continued):

```
private void expandRecursively(MTreeNode node, int n, int k) {  
    List<Integer> seq = node.getSeq();  
    if (seq.size() < k) {  
        for (int i = 1; i <= n; i++) {  
            if (!seq.contains(i)) {  
                List<Integer> newSeq = copyAdd(seq, i);  
                MTreeNode child = new MTreeNode(newSeq);  
                node.getChildren().add(child);  
                expandRecursively(child, n, k);  
            }  
        }  
    }  
}
```

...

MTree ADT: Implementation

- Implementation (continued):

```
private void expandStack(int n, int k) {
    Stack<MTreeNode> stack = new LinkedStack<MTreeNode>();
    stack.push(this.root);
    while (!stack.isEmpty()) {
        MTreeNode node = stack.pop();
        List<Integer> seq = node.getSeq();
        if (seq.size()<k) { for (int i=1; i<=n; i++) { if (!seq.contains(i)) {
            MTreeNode child = new MTreeNode(copyAdd(seq, i));
            node.getChildren().add(child);
            stack.push(child);
        } } } ...
    }
}
```

MTree ADT: Implementation

- Implementation (continued):

```
private void expandQueue(int n, int k) {
    Queue<MTreeNode> queue = new LinkedList<MTreeNode>();
    queue.offer(this.root);
    while (!queue.isEmpty()) {
        MTreeNode node = queue.poll();
        List<Integer> seq = node.getSeq();
        if (seq.size()<k) { for (int i=1; i<=n; i++) { if (!seq.contains(i)) {
            MTreeNode child = new MTreeNode(copyAdd(seq, i));
            node.getChildren().add(child);
            queue.offer(child);
        } } } // DONE!
```

COLLECTION CLASSES & GENERIC PROGRAMMING

Java Collections Framework

- Java comes with a wide library of *collection classes*
- Examples:
 - `ArrayList`
 - `TreeSet`
 - `HashMap`
- idea is to provide well-implemented standard ADTs
- your own ADTs can build upon this foundation
- collection classes store arbitrary objects
- all collection classes implement `Collection` or `Map`
- thus, simple and standardized interface across different classes

Generic Types (revisited)

- type casts for accessing elements are unsafe!
- solution is to use *generic types*
- instead of using an array of objects, use array of some type E
- Example:

```
public class MyArrayList<E> implements List<E> {  
    ...  
    private E[] data;  
    ...  
    public E get(int i) {  
        return this.data[i];  
    }  
}
```


Generic Programming

- the use of generic types is referred to as *generic programming*
- generic types can and should be used:
 - by the user of collection classes
 - Example: `List<String> list = new ArrayList<String>();`
 - when implementing ADTs
 - Example: `public class MyCollection<E> ...`
 - when implementing constructors and methods
 - Example: `public E getElement(int index) { ... }`
 - when implementing static functions
 - Example: `public <E> void add(ListNode<E> n, E elem);`

Generic Programming

- when a class has parameter type `<E>`, `E` is used like normal type
- instances of the class are defined by substituting concrete type
- Example: `public class Mine<E> ... Mine<String> mine = ...`
- more than one parameter is possible
- Example: `public interface Map<K,V> ...`
- when defining static function, prefix return type by parameter `<E>`
- inside function, `E` is used like normal type
- Example: `public <E> void add(ListNode<E> n, E elem);`

Generic Programming

- we can define that a parameter type extends some interface/class

- Example:

```
public interface BinTree<E extends Comparable> { ... }
```

- then all types E are usable, that implement Comparable

- using “?” we can define wildcard types

- Example:

```
public boolean addAll(Collection<? extends E> c) { ... }
```

- here, elements can be any type that extends E

- the same works with “? super E”

Collection ADT: Specification

- interface `Collection<E>` specifies standard operations
 - `boolean isEmpty();` // true, if there are no elements
 - `int size();` // returns number of elements
 - `boolean contains(Object o);` // is object element?
 - `boolean add(E e);` // add an element; true if modified
 - `boolean remove(Object o);` // remove an element
 - `Iterator<E> iterator();` // iterate over all elements
 - `boolean addAll(Collection<? extends E> c);` // add all ...
 - `clear, containsAll, removeAll, retainAll, toArray, ...`
- operations make sense both for lists, queues, stacks, sets, ...
- next: interface `Iterator<E>`

Iterator ADT: Specification

- iterate over elements of collections (= data)
- operations defined by interface `Iterator<E>`:

```
public interface Iterator<E> {  
    public boolean hasNext();           // is there another element?  
    public E next();                   // get next element  
    public void remove();              // remove current element  
}
```

- can be used to access all elements of the collection
- order is determined by specification or implementation

Iterator ADT: Example I

- Example (iterate over all elements of an `ArrayList`):

```
ArrayList<String> list = new ArrayList<String>();
```

```
list.add("Hej");
```

```
list.add("med");
```

```
list.add("dig");
```

```
Iterator<String> iter = list.iterator();
```

```
while (iter.hasNext()) {  
    String str = iter.next();  
    System.out.println(str);  
}
```

- no need to iterate over indices `0, 1, ..., list.size()-1`

Extended for Loop

- also called “for each loop”
- iterative over each element of an array or a collection
- Example 1 (summing elements of an array):

```
int[] numbers = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
int sum = 0;
```

```
for (int n : numbers) {
```

```
    sum += n;
```

```
}
```

- Example 2 (multiplying elements of a list):

```
List<Integer> list = new ArrayList(Arrays.asList(numbers));
```

```
int prod = 1;
```

```
for (int i : list) { prod *= i; }
```

List ADT: Usage

- interface `List<E>` extends `Collection<E>`
- additional operation that make no sense for non-lists (e.g. `get`)
- can be sorted by static method in class `Collections`

- Example:

```
int[] numbers = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
List<Integer> list = new ArrayList(Arrays.asList(numbers));
```

```
Collections.sort(list);
```

- requires that elements implement `Comparable`
- full signature:

```
public static <T extends Comparable<? super T>> void  
    sort(List<T> list);
```


List ADT: Implementations

- **ArrayList** based on dynamic arrays
 - very good first choice in >90% of applications
- **LinkedList** based on doubly-linked lists
 - has prev member variable pointing to previous list node
 - useful when adding and removing a lot in the middle
 - do not use for **Queue** – use **ArrayDeque** instead!
- **Vector** based on dynamic arrays
 - old implementation, not synchronized – use **ArrayList**!
- **Stack** based on Vector
 - do not use for **Stack** – use **ArrayDeque** instead!

Queue ADT: Specification & Implem.

- interface `Queue<E>` extends `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `public boolean offer(E e);` // alternative name to add
 - `public E peek();` // return head
 - `public E element();` // alternative name to peek
 - `public E poll();` // remove and return head
- extended again by interface `Deque<E>` providing support for adding AND removing at both ends
- Implementations:
 - `ArrayDeque` – with `offer == offerLast` and `poll == pollFirst`
 - `LinkedList` – only useful, when not a pure `Queue`

Stack ADT: Specification & Implem.

- class `Stack<E>` implements `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `public E push(E e);` // add on top of stack
 - `public E peek();` // return top element
 - `public E pop();` // remove and return top
 - `public int search(Object o);` // return 1-based index
- superseded by interface `Deque<E>` providing support for adding AND removing at both ends
- Alternative Implementations:
 - `ArrayDeque` – with `push == addFirst` and `pop == removeFirst`

Deque ADT: Specification & Implem.

- interface `Deque<E>` extends `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `addFirst`, `offerFirst`, `addLast`, `offerLast`
 - `removeFirst`, `pollFirst`, `removeLast`, `pollLast`
 - `getFirst`, `peekFirst`, `getLast`, `peekLast`
- `add*`, `remove*`, `get*` throw exceptions
- `offer*`, `poll*`, `peek*` return special value
- Implementations:
 - `ArrayDeque` – fast and preferred
 - `LinkedList` – only use when more than `Deque` needed

Set ADT: Specification

- interface `List<E>` extends `Collection<E>`
- unordered sequences of objects without duplicates
- no additional operations, as `Collection<E>` already specifies
 - `isEmpty`, `size`, `contains`, `add`, `remove`, ...
- no index-based access to elements, as order undefined
- elements MUST implement `equals` and `hashCode` correctly:
 1. for two elements `e1` and `e2` that are equal, both `e1.equals(e2)` and `e2.equals(e1)` must return `true`
 2. for two elements `e1` and `e2` that are equal, we must have `e1.hashCode() == e2.hashCode()`
 3. for two elements `e1` and `e2` that are NOT equal, both `e1.equals(e2)` and `e2.equals(e1)` must return `false`

Set ADT: Example

- Example (intersecting two sets):

```
int[] n1 = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
Set<Integer> set1 = new HashSet<Integer>(Arrays.asList(n1));
```

```
int[] n2 = new int[] {1, 3, 5, 7, 9};
```

```
Set<Integer> set2 = new HashSet<Integer>(Arrays.asList(n2));
```

```
Set<Integer> set3 = new HashSet<Integer>(set1);
```

```
set3.retainAll(set2);
```

- retainAll modifies set3, thus we have (informally):
 - set1 == {1, 2, 3, 5, 7, 11, 13}
 - set2 == {1, 3, 5, 7, 9}
 - set3 == {1, 3, 5, 7}

Iterator ADT: Example 2

- Example (iterate over all elements of a `HashSet`):

```
Set<String> set = new HashSet<String>();
```

```
set.add("Hej");
```

```
set.add("hej");
```

```
set.add("Hej");
```

```
Iterator<String> iter = set.iterator();
```

```
while (iter.hasNext()) {
```

```
    String str = iter.next();
```

```
    System.out.println(str);
```

```
}
```

- prints the two strings in some undefined order

Interface Comparator

- allows to specify how to compare elements

```
public interface Comparator<E> {  
    public int compare(E o1, E o2);    // compare o1 and o2  
    public boolean equals(Object obj); // equals other Comparator?  
}
```

- compare behaves like `o1.compareTo(o2)` from `Comparable<E>`
 - `< 0` for `o1` less than `o2`
 - `==0` for `o1` equals `o2`
 - `> 0` for `o1` greater than `o2`
- `Comparable` defines *natural* ordering
- `Comparator` can define additional orderings

Set ADT: TreeSet Implementation

- `TreeSet` implements sets as special sort trees (Red-Black Trees)
- elements are compared to according to natural ordering
- Example: `public class Compi implements Comparator<Integer> {
 public int compare(Integer i1, Integer i2) {
 return i2.compareTo(i1); }
 public boolean equals(Object other) { return false; } } ...`
`TreeSet<Integer> set1 = new TreeSet<Integer>();
set1.add(23); set1.add(42); set1.add(-3);
for (int n : set1) { System.out.print(" "+n); } // -3 23 42`
`TreeSet<Integer> set2 = new TreeSet<Integer>(new Compi());
set2.addAll(set1);
for (int n : set2) { System.out.print(" "+n); } // 42 23 -3`

Set ADT: Implementations

- **HashSet** based on hash tables
 - very good choice if order really does not matter
- **LinkedHashSet** based on hash tables + linked list
 - in addition to hash table keeps track of insertion order
 - useful for keeping algorithms deterministic
- **TreeSet** based on special sort trees
 - implements the **SortedSet<E>** interface
 - useful for ordered sequences without duplicates
 - can use **Comparators** for different orderings
 - also useful when e.g. hash code not available

Map ADT: Specification

- maps work like dictionaries in Python
- interface `Map<K,V>` specifies standard operations
 - `boolean isEmpty();` // true, if there are no mappings
 - `int size();` // returns number of mappings
 - `boolean containsKey(Object key);` // is key mapped?
 - `boolean containsValue(Object value);` // is value mapped?
 - `V get(Object key);` // return mapped value or null
 - `V put(K key,V value);` // add mapping from key to value
 - `Set<K> keySet();` // set of all keys
 - `Collection<V> values();` // collection of all values
 - `Set<Map.Entry<K,V>> entrySet();` // (key,value) pairs
 - `clear, putAll, remove, ...`

Map ADT: Example

- Example (using and modifying a phone directory):

```
Map<String,Integer> dir = new HashMap<String,Integer>();
dir.put("petersk", 65502327); dir.put("bwillis", 55555555);
for (String key : dir.keySet()) {
    System.out.println(key+" -> "+dir.get(key));
}
for (Map.Entry<String,Integer> entry : dir.entrySet()) {
    System.out.println(entry.getKey()+" -> "+entry.getValue());
    entry.setValue(12345678);
}
dir.keySet().remove("bwillis");
System.out.println(dir);    // only petersk is mapped
```

Hash Table

- a hash table uses the `hashCode` method to map objects to `ints`
- objects are stored in an array
- the position of the object is determined by its hash code modulo the length of the array
- Example: if `o` has hash code `10` and array has length `7`,
`o` is stored at position $10 \% 7 == 3$
- more in **DM507 Algorithms and Data Structures**
- efficient for get and put
- assuming that `hashCode` is implemented in a useful way
- if two or more objects have the same hash code, the array stores a list of objects in that position

Map ADT: Implementations

- **HashMap** based on hash tables
 - very good choice if order does not matter
- **LinkedHashMap** based on hash tables + linked list
 - in addition to hash table keeps track of insertion order
 - useful for keeping algorithms deterministic
- **TreeMap** based on special sort trees
 - implements the **SortedMap<K,V>** interface
 - useful for ordered mappings
 - can use **Comparators** for different orderings
 - also useful when e.g. hash code not available
- **Hashtable** based on hash tables
 - old implementation – only use for synchronization