DM537
Object-Oriented Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

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

ABSTRACT DATATYPES

Abstract Datatype (ADT)

- abstract datatype = data + operations on the data
- **Idea:** encapsulate data + operations with uniform interface

- operations of a datatype
  - at least one constructor
  - modifiers / setters
  - readers / getters
  - computations

- ADTs typically specified by interfaces in Java

Abstract Datatype (ADT)

- abstract datatype = data + operations on the data

- when specifying an ADT, we describe
  - the data and its *logical* organization
  - which operations we want to be able to perform
  - what the results of the operations should be
- we do NOT describe
  - where and how the data is stored
  - how the operations are performed

- ADTs are independent of the implementation (& language)
- one ADT can have many different implementations!

Examples for ADTs

- Numbers: (integer, rational or real)
  - addition, subtraction, multiplication, division, …

- Collections: (collections of elements)
  - List: (ordered collections of elements)
    - Stack (insert & remove elements at one end)
    - Queue (insert at one end, remove at the other)
  - Set: (unordered collection without duplicates)
    - SortedSet (ordered collection without duplicates)
  - Map: (mapping from keys to values)

Developing ADTs

- three steps (like in programming!)

1. specification of an ADT by mathematical means
   - focus on WHAT we want
2. design (still independent of implementation & language)
   - which data structures to use
   - which algorithms to use
   - focus on efficiency of representation and algorithms
   - different data structures give different efficiency for operations
3. implementation (language dependent)
   - select "right" programming language!
   - implement design in that programming language

Specification of an ADT

- mathematically precise!

- data is represented by mathematical objects
- Example: real numbers $\Re$

- operations are mathematical functions
  - explicit specifications
  - Example: $f(x) = x^2$

  - indirect specifications
  - Example: $sqrt : x \in \Re^{\geq 0} \mapsto y \in \Re^{\geq 0}$
    $x = y^2 \wedge y \geq 0$

Integer ADT

- specification:
  - data: all $n \in \mathrm{N}$
  - operations: addition +, subtraction -, negation -, multiplication *, division /
- Design 1: use primitive data type int
  use primitive operations
- Implementation 1: nothing to implement when using Java
- Design 2: use array of bytes to store bit
  provide all relevant operations
- Implementation 2: see class java.math.BigInteger

Integer ADT

- specifying by mathematics often cumbersome
- alternatively use interfaces to specify operations
- alternative specification:
  - data: all $n \in \mathrm{N}$
  - operations:

```
public interface MyInteger {
    public MyInteger add(MyInteger val);      // addition
    public MyInteger sub(MyInteger val);      // subtraction
    public MyInteger neg();                   // negation
    public MyInteger mul(MyInteger val);      // multplication
    public MyInteger div(MyInteger val);      // division
}
```

ABSTRACT DATATYPE FOR LISTS

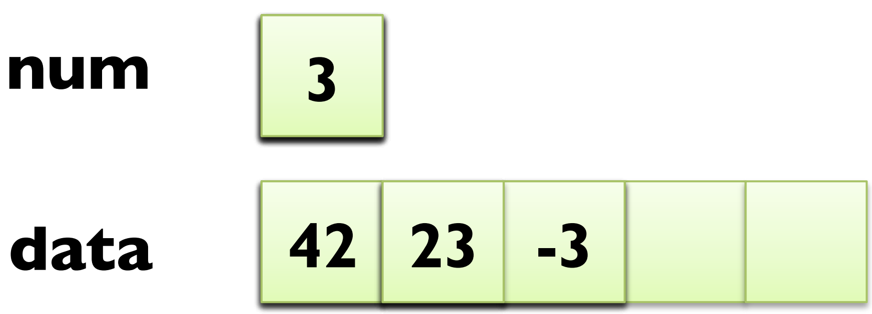
List ADT: Specification

- data are all integers, here represented as primitive int
- operations are defined by the following interface

```
public interface ListOfInt {
    public int get(int i);              // get i-th integer (0-based)
    public void set(int i, int elem);   // set i-th element
    public int size();                  // return length of list
    public void add(int elem);          // add element at end
    public void add(int i, int elem);   // insert element at pos. i
    public void remove(int i);          // remove i-th element
}
```

Partially Full Arrays

- arrays are fixed-length
- lists are variable-length
- **Idea:**
  - use an array of (fixed) length
  - track number of elements in variable
- **Example:** add(23) add(42) add(-3) remove(0) add(1, 23)



List ADT: Design & Implementation 1

- Design 1: partially full arrays of int
- Implementation 1:

```
public class PartialArrayListOfInt implements ListOfInt {
    private int limit;              // maximal number of elements
    private int[] data;             // elements of the list
    private int num = 0;            // current number of elements
    public PartialArrayListOfInt(int limit) {
        this.limit = limit;
        this.data = new int[limit];
    }
    …
}
```

List ADT: Implementation 1

- Implementation 1 (continued):

```
public class PartialArrayListOfInt implements ListOfInt {   …
    private int[] data;
    private int num = 0;   …
    public int get(int i) {
        if (i < 0 || i >= num) {
            throw new IndexOutOfBoundsException();
        }
        return this.data[i];
    }
    …
}
```

List ADT: Implementation 1

- Implementation 1 (continued):

```
public class PartialArrayListOfInt implements ListOfInt {   …
    private int[] data;
    private int num = 0;   …
    public void set(int i, int elem) {
        if (i < 0 || i >= num) {
            throw new IndexOutOfBoundsException();
        }
        this.data[i] = elem;
    }
    …
}
```

List ADT: Implementation 1

- Implementation 1 (continued):

```
public class PartialArrayListOfInt implements ListOfInt {   …
    private int[] data;
    private int num = 0;   …
    public int size() {
        return this.num;
    }
    public void add(int elem) {
        this.add(this.num, elem);          // insert at end
    }
    …
}
```

List ADT: Implementation 1

- Implementation 1 (continued):

```
public class PartialArrayListOfInt implements ListOfInt {   …
    public void add(int i, int elem) {
        if (i < 0 || i > num) {  throw new Index…Exception();  }
        if (num >= limit) {  throw new RuntimeException("full!");  }
        for (int j = num-1; j >= i; j--) {
            this.data[j+1] = this.data[j];   // move elements right
        }
        this.data[i] = elem;                 // insert new element
        num++;                               // one element more!
    }
    …   }
```

List ADT: Implementation 1

- Implementation 1 (continued):

```
public class PartialArrayListOfInt implements ListOfInt {   …
    public void remove(int i) {
        if (i < 0 || i >= num) {  throw new Index…Exception();  }
        for (int j = i; j+1 < num; j++) {
            this.data[j] = this.data[j+1];   // move elements left
        }
        num--;                               // one element less!
    }
    // DONE!
}
```

Dynamic Arrays

- arrays are fixed-length
- lists are variable-length
- **Idea:**
  - use an array of (fixed) length & track number of elements
  - extend array as needed by add method

add(23) add(42) add(-3) add(17) add(31)

- **Example:**

num **5**

data	**23**	**42**	**-3**	**17**	**31**			

List ADT: Design & Implementation 2

- Design 2: dynamic arrays of int
- Implementation 2:

```
public class DynamicArrayListOfInt implements ListOfInt {
    private int limit;              // current maximum number
    private int[] data;             // elements of the list
    private int num = 0;            // current number of elements
    public DynamicArrayListOfInt(int limit) {
        this.limit = limit;
        this.data = new int[limit];
    }
    …
}
```

List ADT: Implementation 2

- Implementation 2 (continued):

```
public void add(int i, int elem) {
    if (i < 0 || i > num) {  throw new Index…Exception();  }
    if (num >= limit) {          // array is full
        int[] newData = new int[2*this.limit];
        for (int j = 0; j < limit; j++) {
            newData[j] = data[j];
        }
        this.data = newData;
        this.limit *= 2;
    }
    …    }      // rest of add method
```

List ADT: Design 2 Revisited

- Design 2 (revisited): symmetric dynamic arrays of int
  - keep startIndex and endIndex of used indices
  - start with startIndex = endIndex = limit / 2
  - i.e., limit / 2 free positions at the beginning
  - i.e., limit / 2 free positions at the end
  - extend array at the beginning when startIndex < 0 needed
  - extend array at the end when endIndex > limit needed
  - shrink array in remove, when (endIndex – startIndex) < limit / 4

List ADT: Design 3

- goal is to use list for arbitrary data types
- Design 3: dynamic arrays of objects
- Implementation 3:

How to use with int, double etc.?

```
public class DynamicArrayList imple… List {
    private int limit;            // current maximum number
    private Object[] data;        // elements of the list
    private int num = 0;          // current number of elements
    public DynamicArrayListOfInt(int limit) {
        this.limit = limit;
        this.data = new Object[limit];
    } …
}
```

Boxing and Unboxing

- primitive types like int, double, … are not objects!

- Java provides wrapper classes Integer, Double, …
- Example:
```
Integer myInteger = new Integer(13);
int myInt = myInteger.intValue();
```

- transparent due to *automatic boxing* and *unboxing*
- Example:
```
Integer myInteger = 13;
int myInt = myInteger;
```

- useful when e.g. storing int values in a Object[]

List ADT: ArrayList

- Java provides pre-defined symmetric dynamic array list implementation in class java.util.ArrayList
- Example:

```
ArrayList myList = new ArrayList(10);          // initial limit 10
for (int i = 0;  i < 100;  i++) {
    myList.add(i*i);                          // list of squares of 0 … 99
}
System.out.println(myList);
for (int i = 99;  i >= 0;  i--) {
    int n = (Integer) myList.get(i);          // get returns Object
    myList.set(i, n*n);                       // now to the power of 4!
}
```

Generic Types

- 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];
    }
}
```

Finding in Lists

- finding typical example for another List ADT operation
- specified by the following method signature:

```
public int indexOf(E elem) {
    for (int i = 0; i < this.size(); i++) {
        E cand = this.get(i);
        if (elem == null ? cand == null : elem.equals(cand)) {
            return i;          // found an equal element
        }
    }
    return -1;                 // did not find any match
}
```

Sorting Lists

- sorting is another important List ADT operation
- many different approaches to sorting exist
- more on this: **DM507 Algorithms and Data Structures**
- Example (Selection Sort):

```
private void swap(int i1, i2) {
    E temp = this.get(i1);
    this.set(i1, this.get(i2));
    this.set(i2, temp);
}
```

num | **8** | **42**

this.swap(1,3)

data							
23	**17**	**-3**	**42**	**31**	**97**	**71**	**59**

Sorting Lists

- sorting is another important List ADT operation
- many different approaches to sorting exist
- more about this in DM507 Algorithms and Data Structures
- Example (Selection Sort):

```
public void selectionSort() {
    for (int firstPos = 0;  firstPos < this.size()-1;  firstPos++) {
        int minPos = this.size()-1;  // assume last element is smallest
        for (int i = firstPos; i < this.size()-1; i++) {
            if (this.get(i) < this.get(minPos)) {  minPos = i;  }
        }
        this.swap(minPos, firstPos);
} }
```

Sorting Lists

```
public void selectionSort() {
   for (int firstPos = 0;  firstPos < this.size()-1;  firstPos++) {
      int minPos = this.size()-1;  // assume last element is smallest
      for (int i = firstPos; i < this.size()-1; i++) {
         if (this.get(i) < this.get(minPos)) {  minPos = i;  }
      }
      this.swap(minPos, firstPos);
   }
}
```

num	8	firstPos	6	minPos	6

data	-3	17	23	31	42	59	71	97