Introduction to Programming

René Thiemann

Institute of Computer Science
University of Innsbruck

WS 2008/2009

Outline

Outline

Capital growth revisited

- capital after n years $= \begin{cases} \text{capital} & \text{if } n = 0 \\ \frac{100+\text{rate}}{100} \cdot \text{capital after } n-1 \text{ years} & \text{otherwise} \end{cases}$
- implementation using loops:

```
double cGrowth(int n, double cap, double rate) {
  for (int i = 0; i < n; i++)
    cap *= (100+rate)/100;
  return cap;
}
```

- implementation with recursion:

```
double cGrowth(int n, double cap, double rate) {
  if (n == 0)
    return cap;
  else
    return (100+rate)/100 * cGrowth(n-1, cap, rate);
}
```

Divide & Conquer

- idea of divide & conquer
  - solve small problems directly

$$f(S) = result$$

  - divide large problem in (several) small ones and solve these
  - assemble solution of large problem from small solutions

$$L = S_1 \cup \cdots \cup S_n \qquad f(L) = assemble(f(S_1), \ldots, f(S_n))$$

- often problems can easily be solved using divide & conquer-algorithms
- example: sorting by merging (ms)
  - $ms([]) = []$ $\quad ms([e]) = [e]$
  - $ms([e_1, \ldots, e_n]) = merge(ms([e_1, \ldots, e_{\frac{n}{2}}]), ms([e_{\frac{n}{2}+1}, \ldots, e_n])$, if $n \geqslant 2$
  - $merge$ takes two sorted arrays and returns one new sorted array
- divide & conquer directly possible with recursion
- implementation via loops often difficult

Recursion

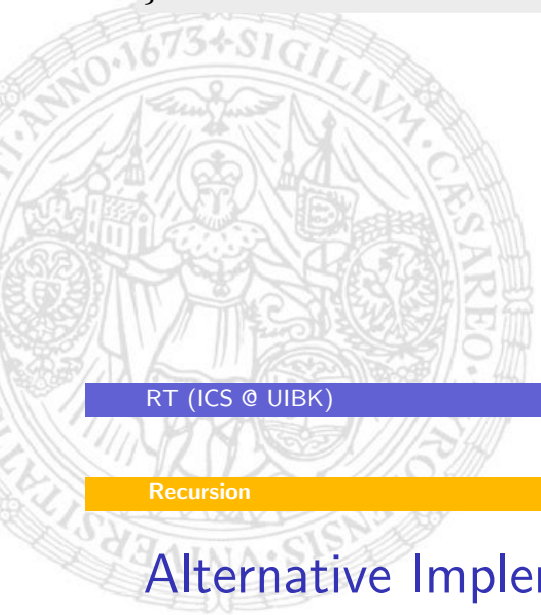
a method is recursive iff it leads to a call to itself

- direct recursion: second cGrowth-example, merge-sort
- indirect recursion:

```
boolean even(int n) {
  if (n == 0)
    return true;
  else
    return odd(n-1);
}
boolean odd(int n) {
  if (n == 0)
    return false;
  else
    return even(n-1);
}
```

Visualization

```
double cGrowth(int n, double cap, double rate) {
  if (n == 0)
    return cap;
  else
    return (100+rate)/100 * cGrowth(n-1, cap, rate);
}
```



Alternative Implementation

```
double cGrowth(int n, double cap, double rate) {
  if (n == 0)
    return cap;
  else
    return cGrowth(n-1, (100+rate)/100*cap, rate);
}
```

n	2	
cap	10000.00	
rate	4.5	
result	10920.25	= recursive result
n	1	
cap	10450.00	
rate	4.5	
result	10920.25	= recursive result
n	0	
cap	10920.25	
rate	4.5	
result	10920.25	

Non-linear Recursion

- linear recursion: at most one recursive call (cGrowth, even-odd)
- non-linear recursion: more than one recursive call (merge-sort)

```
void hanoi(int n, String from, String via, String to) {
    if (n > 0) {
        hanoi(n-1, from, to, via);
        println(from + " -> " + to);
        hanoi(n-1, via, from, to);
    }
}
```

- sometimes non-linear recursion required
- use non-linear recursion with care, easy to obtain inefficient code
  - merge-sort: good efficiency
  - hanoi: inefficient, but cannot be avoided (solution is optimal)
  - fibonacci: see next slide

Fibonacci-numbers

stairs with n-steps; person can take one or two steps at a time
question: how many possibilities are there to walk down the stairs?

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

- direct implementation: inefficient, exponential complexity

```
int fib(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

  problem of overlapping calls: `fib(n)` calls `fib(n-2)` twice
- here: better use bottom-up computation with linear recursion

⇒ linear complexity

Summary

- divide & conquer is a general programming pattern
  - solve trivial problems directly
  - divide large problems into simpler ones, solve these, and assemble solution from partial solutions
- a method can call itself recursively

⇒ alternative to loops

- different forms of recursion
  - direct/indirect
  - linear/non-linear
- non-linear recursion should be used with care (problem if smaller inputs are overlapping)

Outline

- Recursion
- Constructors
- Recursive/Dynamic Data Structures

Creating objects

- currently: creating objects and initializing values is done in two steps

```
Person p = new Person(); // creation
p.setFirstName("John"); // init values
p.setLastName("Wood");
```

- idea of constructors: creation and initialization in one step
- constructors: special kinds of methods
  - name of constructor-method = name of class
  - no return type in declaration (is always a new object of that class)
  - when called, a new object is created
  - syntax: `private/public ClassName(type1 arg1, type2 arg2 ,...){...}`

```
public Person(String firstName, String lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
```

  - calling a constructor: `new ClassName(expr1, expr2, ...)`
  - above code becomes `Person p = new Person("John", "Wood");`

Multiple Constructors

- often several constructors are offered

```
public Person(String firstName, String lastName) {...}
public Person(String fullName) {...}
public Person(String fullName, double salary) {...}
```

⇒ several methods with the same name (overloading)

- when calling such a method/constructor the most suitable is chosen

```
new Person("John", "Wood");
new Person("John Wood");
new Person("John Wood", 10000);
```

⇒ different methods with same name require distinct parameter-types!

`public Person(String lastName, String firstName) {...}` not allowed

- constructors can call each other via `this(expr1 ,...)` in first line

```
public Person(String fullName, int salary) {
  this(fullName);
  this.salary = salary;
}
```


Outline

The Need for Dynamic Data Structures

- currently: to store several persons we can use arrays
- reminder: arrays have a fixed (static) size
- adding a new person might require increased array-size

⇒ need to create new larger array and copy all persons from old array into new array

⇒ not optimal

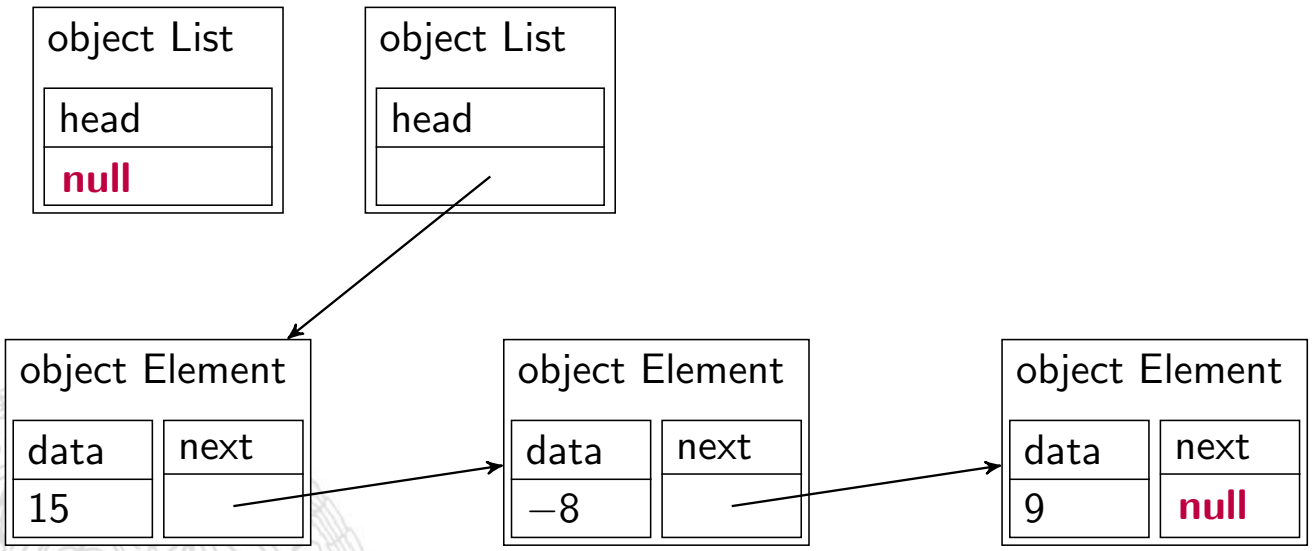
- desire: data-structure which can store multiple persons and be expanded easily
- solution: dynamic data structures

Dynamic = Recursive Data Structures

- recursive method: method which calls itself
- recursive data-structure: data-structure which contains itself

```
class List {
    Element head;
}
class Element {
    int data;  // could also be a person
    Element next; // recursive
}
```

Lists with zero and three elements



Working with lists

Implementation 1

```
public class List {
    private Element head;
    public List() {
        this.head = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void insert(int data) {
        head = new Element(data, head);
    }
}
class Element {
    Element(int data, Element next) {
        this.data = data;
        this.next = next;
    }
}
```


Implementation 2

```
public class List {
    public int size() {
        if (head == null) {
            return 0;
        } else {
            return head.size();
        }
    }
}
class Element {
    int size() {
        if (next == null) {
            return 1;
        } else {
            return 1+next.size();
        }
    }
}
```

```
public class List {
    public String toString() {
        return this.toString(false);
    }
    public String toString(boolean reverse) {
        String result = "(";
        if (head != null)
            result += head.toString(reverse);
        return result + ")";
    }
}
class Element {
    String toString(boolean reverse) {
        if (next == null) {
            return data+"";
        } else {
            String res = next.toString(reverse);
            if (reverse)
                return res+","+data;
            else
                return data+","+res;
        }
    }
}
```


Implementation 4

```
public class List {
    public boolean contains(int data) {
        return head != null && head.contains(data);
    }
}
class Element {
    boolean contains(int data) {
        if (this.data == data) {
            return true;
        }
        if (next == null) {
            return false;
        } else {
            return next.contains(data);
        }
    }
}
```

Implementation 5

```
public class List {
    public void insertSorted(int data) {
        head = insertSorted(data, head);
    }
    private static Element insertSorted(int data, Element e) {
        if (e == null) {
            return new Element(data, null);
        } else if (data < e.getData()) {
            return new Element(data, e);
        } else {
            e.setNext(insertSorted(data, e.getNext()));
            return e;
        }
    }
}
```


Implementation 6

```
public class List {
    public void removeAll() {
        head = null;
    }
    public int removeHead() {
        int res;
        if (head == null) {
            println("error, list is empty");
            res = (int)Math.random();
        } else {
            res = head.getData();
            head = head.getNext();
        }
        return res;
    }
}
```

Implementation 7

```
public class List {
    public void remove(int data) {
        if (head != null) {
            head = head.remove(data);
        }
    }
}
class Element {
    Element remove(int data) {
        if (this.data == data) {
            return next;
        } else {
            if (next != null) {
                next = next.remove(data);
            }
            return this;
        }
    }
}
```


Other important recursive data-structures

- list with direct access on both sides

```
public class List {
    private Element head, tail;
    public void insertEnd(int data) {...};
}
```

- binary tree

```
public class BinTree {
  private Node root;
}
class Node {
  private Node left, right;
  int data;
}
```

- ...

Summary

- recursive data-structures contain multiple data and feature dynamic size-adjustment

$\Rightarrow$ number of elements need not be fixed at creation time

- implementation via two classes:
  - some public class which provides public methods
  - some non-public element class which contains attribute(s) of element class (recursively)

- internal methods are often recursive and manipulate references
- external calls don't have to care about these internals and should use public methods of public class as black-box!