

# Imperative Programming

Slide Series 1

# Content

Object orientation/object oriented programming/UML

Imperative programming and state

By value/By reference

Side effects/Referential transparency

Imperative/Declarative style

Declarative style in imperative programs

Correctness

Testing

Linked data structures

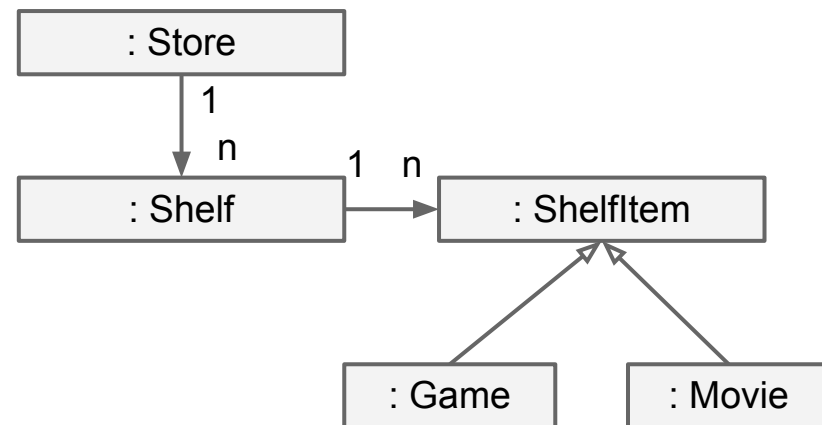
# The Object Oriented Paradigm

Program seen as a collection of interacting objects

- An "**object model**" of some problem
- Close connection between problem and program
- No object model => Not an object oriented program

# UML

Unified Modelling Language, graphical language to describe different aspects of an OO-model



UML Class diagram

# UML, cont

Possible to use UML in formal ways, but we don't

We use it as a shorthand, to communicate principles and ideas at a higher level (than code)

- Using a very small subset of symbols (mostly from class diagrams)

# Object Oriented Programming

Using programming techniques designed to support creation and execution of object models

Programming techniques include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance

# Imperative Programming

Imperative programming is a programming paradigm that describes computation in terms of **statements** that change a program **state**

// Wikipedia

- Describes **how** thing should be done!
- Contrast: **Declarative programming**, **what** to be accomplished (Functional programming, Logic programming, ...)

# State

We do imperative OO programming (in Java)

- We have an object model
- We have objects
- Objects have state

**State** = all the stored information, at a given point in time (in this course: the values for all non-local variables at any time)



# Variables

In functional programming variables bound to values (will never change)

In imperative programming variables (attributes) are names of memory location, like boxes (will change content)

# Statements

Imperative programs are built up by (sequences of) statements (in Java terminated by ;)

- The smallest standalone element
  - Example simple stmts: **return; System.out.println();**
  - Example compound: **if, switch, while, for**
  - Denotes an action (not a value)
- The sequence (ordering) often matters

No statements in (pure) functional programming, no ordering concerns (can use do-notation in Haskell)

# Expressions

Statement built up by expression

- An expression denote a value

```
// Expressions
true && false || 1 > 0;
1 + 4.5
"hej".length();
x = y = 1;    // Part of statement
```

In Java all values have types ... so an expression...

# State is Problematic

Put simple: State means we have many, many, ... boxes

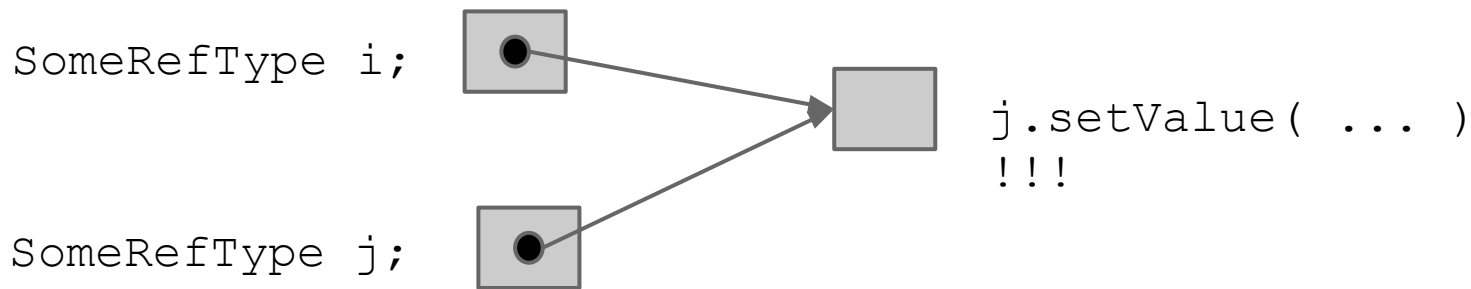
- Boxes hold the (partial) result and information how to proceed with the calculation
- During execution the content of the boxes will change
- Have to put correct value of the correct type in the correct box in the correct order...

... this is **very** hard in any non-trivial application.

- If any mistake program is in **invalid state**

# In Fact it's Even Worse

The boxes can have references to other boxes (with references to yet other boxes ...)



We possible have **shared state** (alias problem)

*"If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object, and then the altered state can be observed through the reference in the other variable." // JLS 4.3.1*

# Value and Reference Semantics\*

The possibility of "reference or not" have impact on the semantics (the meaning)

Value semantics (**by value**)

- No shared state (copies creates)

Reference semantics (**by reference**)

- Shared state

*This slide is special.. why???*

**Recurring question:** Is this by value or by reference?

# Example: By Value or By Reference

```
Integer i = new Integer(4)
Integer j = new Integer(4)
if( i == j ){
}
```

**False** by reference  
semantics

```
int i = 4
int j = 4
if( i == j ){
}
```

**True** by value semantics

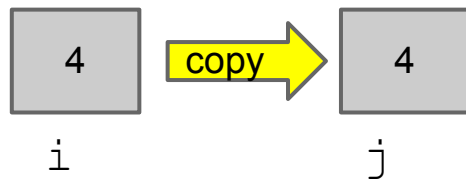
How about this?

```
if ( i <= j ){
}
```

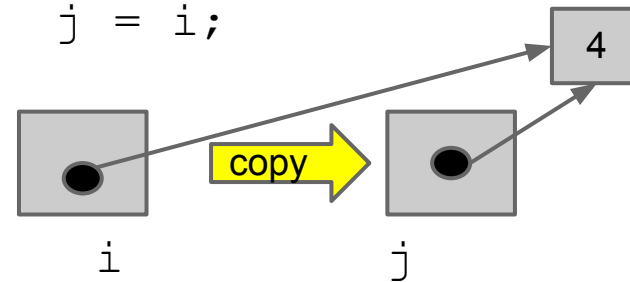
# Call by Value

Java have references but all calls (and assignments) are by value i.e. a value is copied from a variable to another

```
int i = 4;  
int j;  
// After this we have 2 4's  
j = i;
```



```
Integer i = 4;  
Integer j;  
//After this we have 2  
//references  
j = i;
```



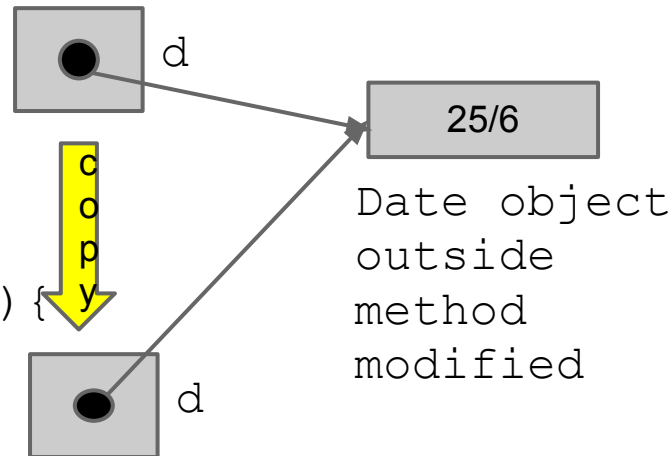


# Call by Value, cont

Call by value of methods have implications

```
Date d = new Date();  
o. doIt( d )
```

```
public void doIt( Date d ){  
    d.set(...);  
}
```



# Out Parameters\*

Normally method parameters should be "read only"

- If not, document (i.e. "Call will modify argument...")
- Confusing if caller uses reference after method call, invisible state change
- If need more return values, create object to return (don't use outparam as return)

... ok usage

- Passing an array/Collection to be manipulated by method is ok

# Side Effects

Side effect = In addition to compute a value something more happens (state modified)

Heavy use of side effects in imperative programming

```
List<Integer> is = ...  
// Add a value and modify list  
boolean b = is.add(123);  
  
// Assignment causes side effect  
x = 1;
```

# Referential Transparency\*

```
// Always 0 in functional programming  
putStrLn( f(123) - f(123) )
```

```
// In imperative OO ???  
System.out.println(o.f(123) - o.f(123))
```

Imperative languages are not referentially transparent (i. e. not always same result for same argument)

- Because of possible side effects
- Makes it hard to reason about imperative code

# Mutator and Accessors\*

Make explicit which methods have side effects

## Accessors-method (getters)

- Never change state of object, no side effects
- Used to retrieve information (state or calculated)
- ... more later

## Mutator-method (setters)

- Changes state of object
- Don't return information about object (but possible other result, boolean common)

# Imperative Style vs Declarative\*

```
//Naive Haskell power function (declarative)
```

```
pow a 0 = 1
```

```
pow a b = a * pow a (b-1)
```

Like an equation.  
How to prove correctness of this?

```
// Same in imperative style
```

```
public int pow( int a, int b){
```

```
    int result = 1; int i = 0; // Bad style
```

```
    while( i < b ){
```

```
        result *= a;
```

```
        i++;
```

```
    }
```

```
    return result;
```

```
}
```

Describe step by step.  
How to prove correctness of this?

# Declarative Style Proof

Prove:  $\text{pow } a \ b = a^b$

By induction

- Base:  $\text{pow } a \ 0 = 1 = a^0 \quad (b = 0)$
- Assume:  $\text{pow } a \ b = a^b \quad (b > 0)$

Show:  $\text{pow } a \ b+1 = a^{b+1}$

$$\begin{aligned} \text{pow } a \ b+1 &= a * \text{pow } a \ ((b+1) - 1) = \\ &= a * \text{pow } a \ b = a * a^b = a^{b+1} \end{aligned}$$

# Imperative Style Proof

Prove:  $\text{pow}(a, b) = a^b$

By use of loop **invariants** (boolean expressions)

- Show invariant is true prior to first iteration
- If it's true before an iteration show it's true before next
- At termination deduce result (or something stronger) from (the true) invariant (an implication)



# Imperative Style Proof, cont

Invariant:  $i \leq b \ \&\& \ result == a^i$

// If  $b == 0$  trivially true, assume  $b > 0$

```
public int pow( int a, int b){
```

```
    int result = 1; int i = 0;
```

```
    while( i < b ){
```

```
        result *= a;
```

```
        i++;
```

```
    }
```

```
    return result;
```

```
}
```

Invariant true prior to first iteration

$i < b \ \&\& \ i == 0 \ \&\& \ result == 1 \Rightarrow$   
 $i \leq b \ \&\& \ result == a^i$

Assume invariant true before iteration.

$result *= a; \Rightarrow result == a^{(i+1)}$  (inv. false )

$i++;$  (inv. true)

Invariant true after loop

At termination (invariant still holds):  $i \leq b \ \&\& \ result == a^i \ \&\& \ !(i < b)$   
 $\Rightarrow i == b \ \&\& \ result == a^i \Rightarrow result == a^b$

# Hard Parts of Imperative Proofs

The negation of the guard ( $i \geq b$ ) and the invariant should imply ( $\Rightarrow$ ) the desired outcome ( $\text{result} == a^b$ )

- What's is the desired outcome (easy in this case)?
- How to find invariant
- How to keep invariant but eventually terminate the loop (how to eventually get the guard false)?

# Limitations of Imperative Style Proof

No side effects

- No instance variables

Must have single entry and exit point

- No **break**, **continue** or **return**

And more...

# Declarative Style in Imperative Language

```
// Java going declarative
public int powR( int a, int b){
    if( b == 0){
        return 1;
    }else{
        return a * powR( a, b-1);
    }
}
```

Possible but ...

- Watch out for StackOverflowException
- Should prefer tail recursion (powR not tail recursive)
- Even so; Tail call optimization possible not supported (depends on JVM/JIT)

# Tail Recursion\*

```
// Tail recursive version of pow, must init result to 1
public int powTail( int a, int b, int result){
    if( b == 0 ){
        return result;
    }else{
        // Tail recursive, nothing to do after call returns
        return powTail( a, b-1, result * a);
    }
}
```

If tail call optimization, this should run as fast as imperative version and no StackOverflowException


Accumulator parameter holding the result

# Tail Recursion to Imperative Style\*

Tail recursion easy to convert to imperative loop

We run  $h(x)$  possible many times and finally  $g(x)$  on result

```
public f(x) {  
    if (p(x)) {  
        return g(x);  
    } else {  
        return f(h(x));  
    }  
}
```



```
public f(x) {  
    while (!p(x)) {  
        x = h(x);  
    }  
    return g(x);  
}
```

If many base cases => negate disjunction of base cases

# Proving Tail Recursion

As demonstrated tail recursion is a loop...

... so sadly have to use invariants for the accumulator parameter ....

# Proof vs Reasoning

Imperative proofs quickly becomes very complicated ... (one example later)

... but using the techniques for **reasoning** is useful

- Will gain understanding
- Will improve code
- If completely impossible to imagine any kind of proof when inspecting the code ... rework it!
- Use natural language for reasoning



# Testing

Proving difficult, tedious, ...

Reasoning informally using proof techniques possible...

...other attempt: Testing...

"program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."//E. Dijkstra

# Unit Testing

Testing the smallest units (parts) of the program

- I.e. we test classes

One test for each class, testing all public methods

- If method private, normally not in test. If in need, change to public and back (bad.., just for now)
- If method void, need to inspect state (possible extra method `getNNN()`)

# JUnit\*

## Test framework for Java

- We have a class (class under test, CUT)
- Write another "test"-class, testing all (public) methods in CUT
  - Test class usually has one test method for each method in CUT
- Let JUnit run the test class
- JUnit will report any failures

## JUnit part of Eclipse (a plugin)

- Separate test code (test classes), use a test-source folder in Eclipse project

# Organizing Test Code

Always keep test code separate from application code

- Use source folder "test" in Eclipse
- Use same package structure as application, more to come...

# Code Coverage

*"Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested." //Wikipedia*

Possible to see how much of code is run during tests, high percentage good (90% or more)

Many tools, ECLEmma a plugin for Eclipse

# Other Benefits of Testing

Confidence in doing changes, just re-run the tests!

- All passed tests should pass after any modification

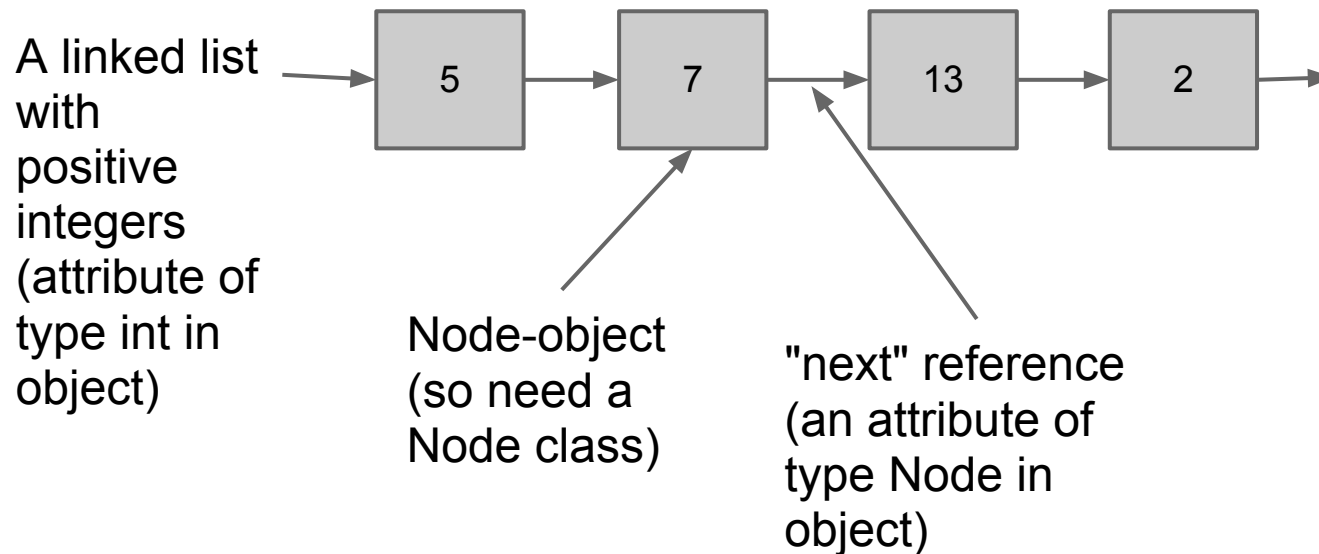
Documentation

- Use very descriptive names for test methods

NOTE: Testing is a complicated "art", we'll just scratch the surface

# Linked Data Structures

We'll use a lot of linked structures (i.e. collections of objects connected by references)



# The Node Class

```
// Class for objects used in a linked data structure
public class Node {
    private int data;
    private Node next;  // The next reference

    // Better to set data also, this is just for now
    public Node(Node next){
        this.next = next;
    }
    // set/get methods
}
```

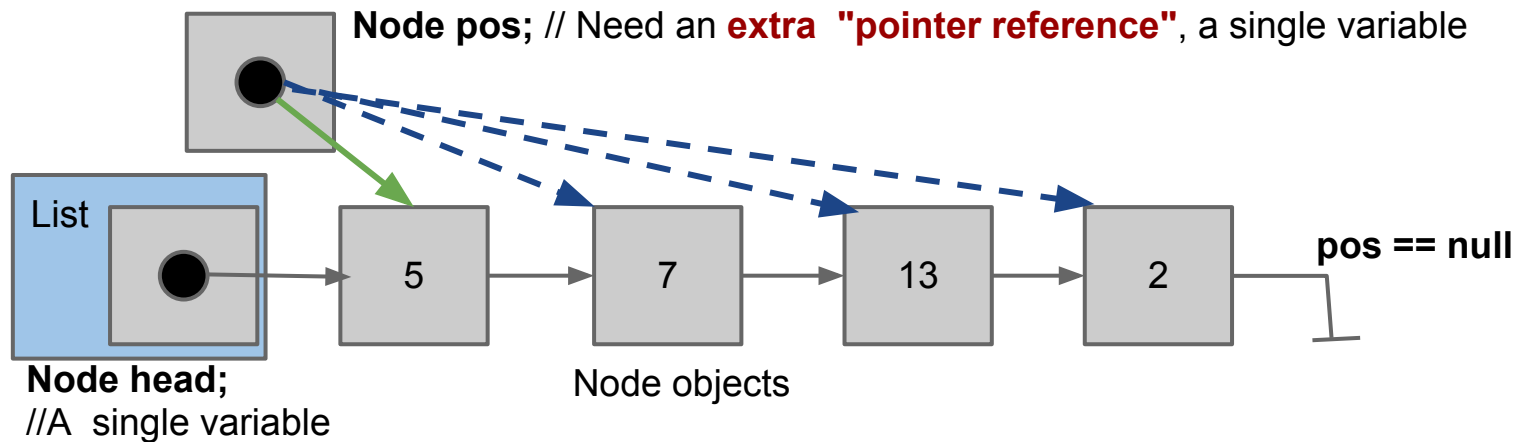


# The List Class

```
// Class for managing the linked Node-structure
public class List {
    // Reference to the first node, all we need
    private Node head;

    public void add(){
        Node n = new Node(head);
        head = n;
    }
}
```

# Traversing a Linked List\*



Can't change head  
variable, if so whole  
structure lost

```
pos = head; // Init pointer
while( pos != null){
    // Do something with node
    // Move pointer to next
    pos = pos.next;
}
```

# An Initialization Method

```
// Using classes from previous slides
public class List {
    ...
    public void init() {
        Node pos = this.head;
        while (pos != null) {
            pos.setData(1);
            pos = pos.getNext();
        }
    }
}
```

How to prove this correct (at termination all nodes have data == 1)?

# Proof of Init

We must prove for all nodes  $\text{data} == 1$

Hard to find loop invariant (we assume list is not circular)..?

We use the reachability function  $R_{\text{next}}(u)$

- a function returning a set of nodes reachable from  $u$ .

Axiom ( $v$  and  $u$  are nodes):

$$v \in R_{\text{next}}(u) \Leftrightarrow (v == u \parallel (u.\text{next} != \text{null}) \ \&\& \ v \in R_{\text{next}}(u.\text{next}))$$

Assume  $\text{null}.\text{next} = \text{null}$  (no loss of generality)

# Proof of Init, cont

Invariant:  $\forall v \in R_{\text{next}}(\text{head}) : (\text{pos} \neq \text{null} \ \&\& \ v \in R_{\text{next}}(\text{pos})) \ || \ v.\text{data} == 1$

For all nodes v in list; it's possible to reach v via pos or v.data == 1

```
public void init() {
    Node pos = this.head;
    while (pos != null) {
        pos.setData(1);
        pos = pos.getNext();
    }
}
```

True  
before  
loop

## First iteration

pos != null && v in R(pos) is true  
pos.data != 1 (but it's an disjunction so true)

## Sec. iteration

pos != null && v in R(pos) or pos.data == 1 true  
(can't reach first but it has data set to 1)

If pos == null inv. still true!

At loop termination (define P(pos) as pos != null, Q(pos) as  $v \in R_{\text{next}}(\text{pos})$ ). We have;

**!P(pos)** &&  $\forall v \in R_{\text{next}}(\text{head}) : ((\text{P(pos)} \ \&\& \ \text{Q(pos)}) \ || \ v.\text{data} == 1) \Rightarrow$

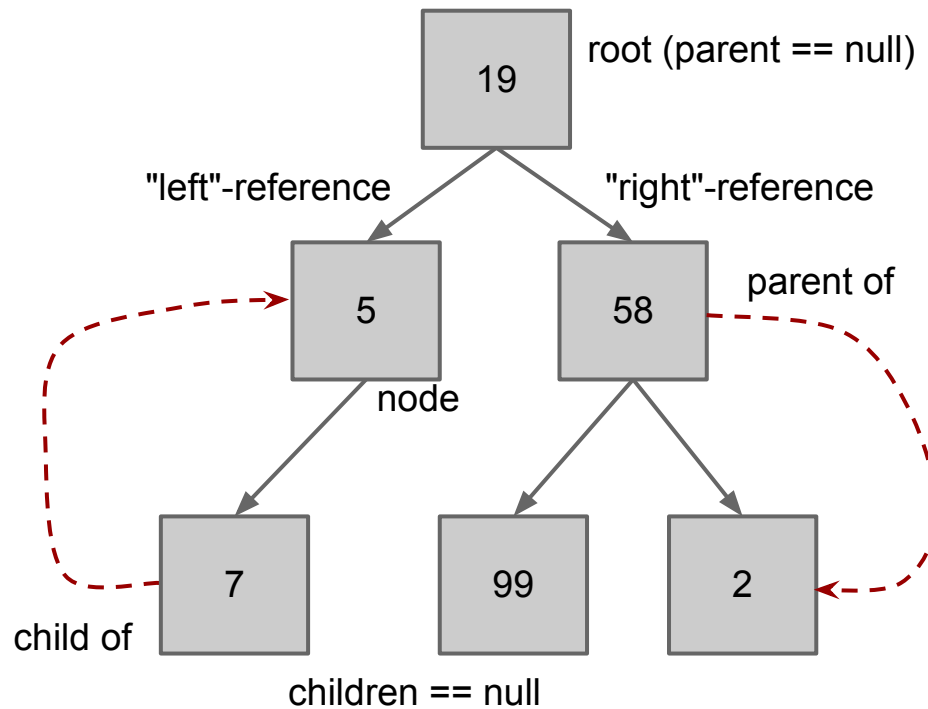
$\forall v \in R_{\text{next}}(\text{head}) : (\text{!P(pos)} \ \&\& \ (\text{P(pos)} \ \&\& \ \text{Q(pos)}) \ || \ v.\text{data} == 1) \Rightarrow$

$\forall v \in R_{\text{next}}(\text{head}) : (\text{false} \ \&\& \ \text{Q(pos)} \ || \ v.\text{data} == 1) \Rightarrow \forall v \in R_{\text{next}}(\text{head}) : (\text{false} \ || \ v.\text{data} == 1)$

$\forall v \in R_{\text{next}}(\text{head}) : v.\text{data} == 1$

# Trees\*

Another linked data structure



# A Node Class for Trees\*

```
// Class for nodes in a tree
public class Node {
    // References to other nodes in tree
    private Node parent;
    private Node left;
    private Node right;

    // set/get methods

}
```

# Count Nodes in Tree\*

```
// In Tree class
public int countNodes() { // Method to get going
    return countNodesR(root);
}

// Declarative style counting nodes (imperative hard...).
private int countNodesR(Node node) {
    if( node == null ) {
        return 0;
    } else {
        return 1 + countNodesR(node.left) +
                                countNodesR(node.
right);
    }
}
```



# Summary

- We are doing imperative OO-programming which implies state (and statements)
- State is very complex
- References makes it even harder (different semantics)
- Proving imperative programs is hard
  - Declarative more natural
- We try to reason informally using ideas from the presented proof techniques
- An alternative to proofs is testing
- Linked data structures are collections of connected objects