Imperative Object Oriented 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*, modeling language to describe different aspects of an OO-model (graphic notation techniques)

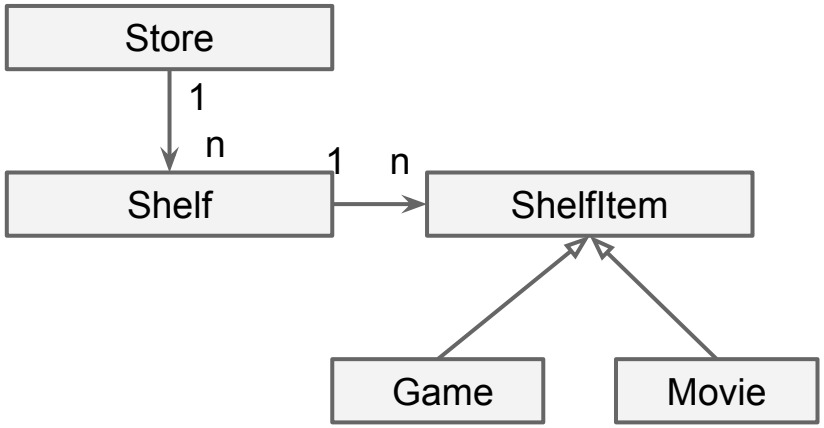
Many kind of visual diagrams

- Class diagram (we only use this)
- Sequence diagram
- ...

*) See examples of UML diagrams far down right

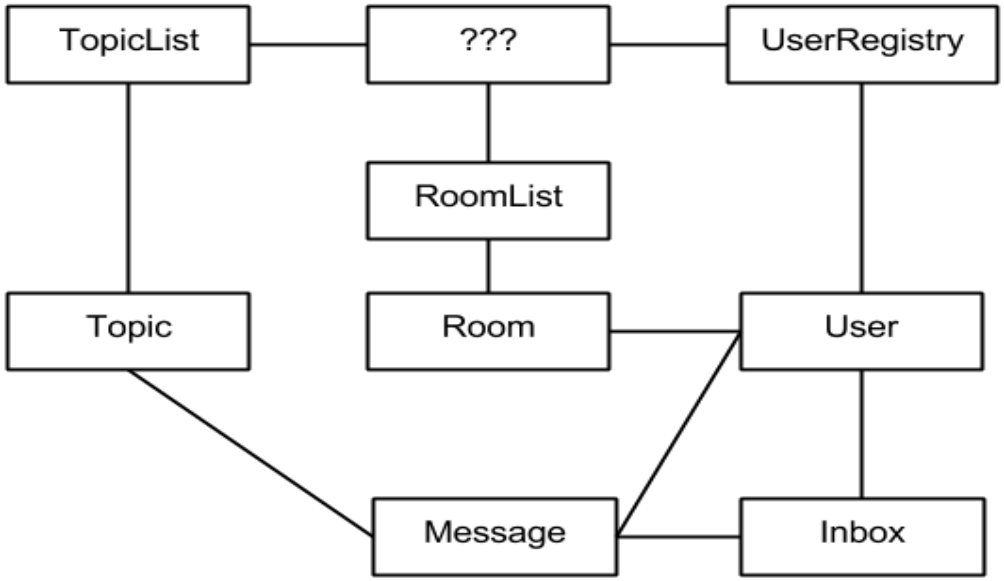
UML Class Diagram

Used to show structure. Classes as rectangles, references (between objects) as lines, … confusing! Number of objects referenced indicated



UML Class diagram of store

A Model of … ?


TopicList
???
UserRegistry
RoomList
Topic
Room
User
Message
Inbox

Usage of UML

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

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

- Using a very small subset of symbols (mostly from class diagrams)
- More to come...

Object Oriented Programming

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

Programming techniques include features such as classes and/or objects, 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 instance variables at any time (for all objects))

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 stmts: **if, switch, while, for**
  - Denotes an action (not a value)
  - Similar to a sentence in natural language
- The sequence ordering often matters

Expressions

Statement built up by expression

- An expression denote a value, composed of literals (explicit values in code), variables, method calls, operators, ...

```
// Expressions
true && false || 1 > 0
1 + 4.5             // 1 and 4.5 are literals
"hej".length()
x = y = 1           // What's the value?
```

Variables

Local variable (incl. parameters/arguments to methods)
- Variable declared inside method (also in parameter list)

Instance variable aka **attribute** or **field**
- Variable declared in class outside any method

"A declaration introduces an entity into a program and includes an identifier (§3.8) that can be used in a name to refer to this entity." // JLS 6.1

Variables and Memory

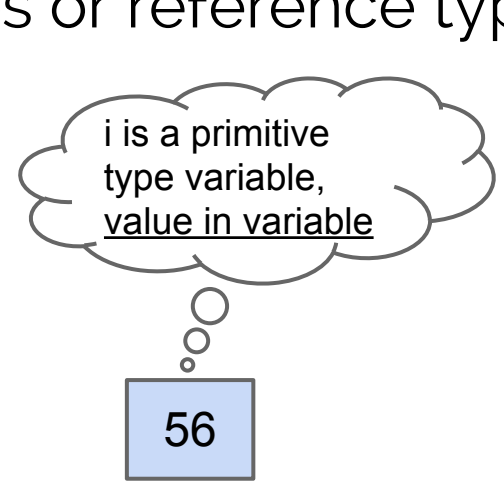
All variables exists in memory but local variables are technically different from instance variables

- Local variables are located in an area called **the stack**. This area is reused by all methods. Local variables only exist during the execution of the method

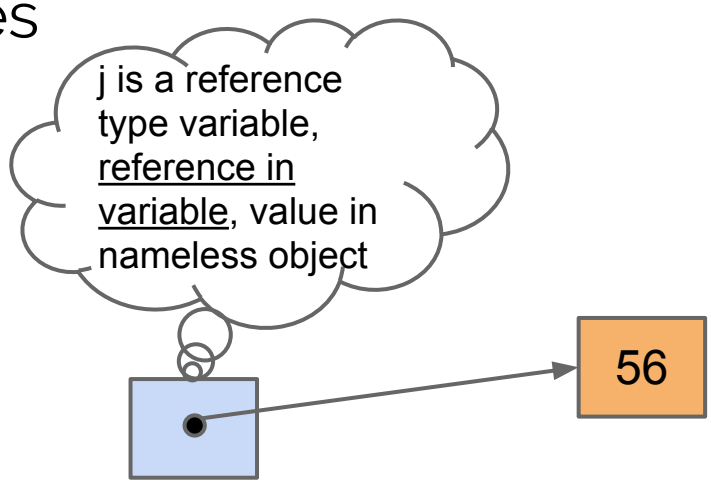
- Instance variables (and the objects) are located in a memory area called **the heap**, will exist as long as the object exists

Primitive vs Reference Variable

Two very different kind of variables, holding primitive types or reference types



int i = 56;

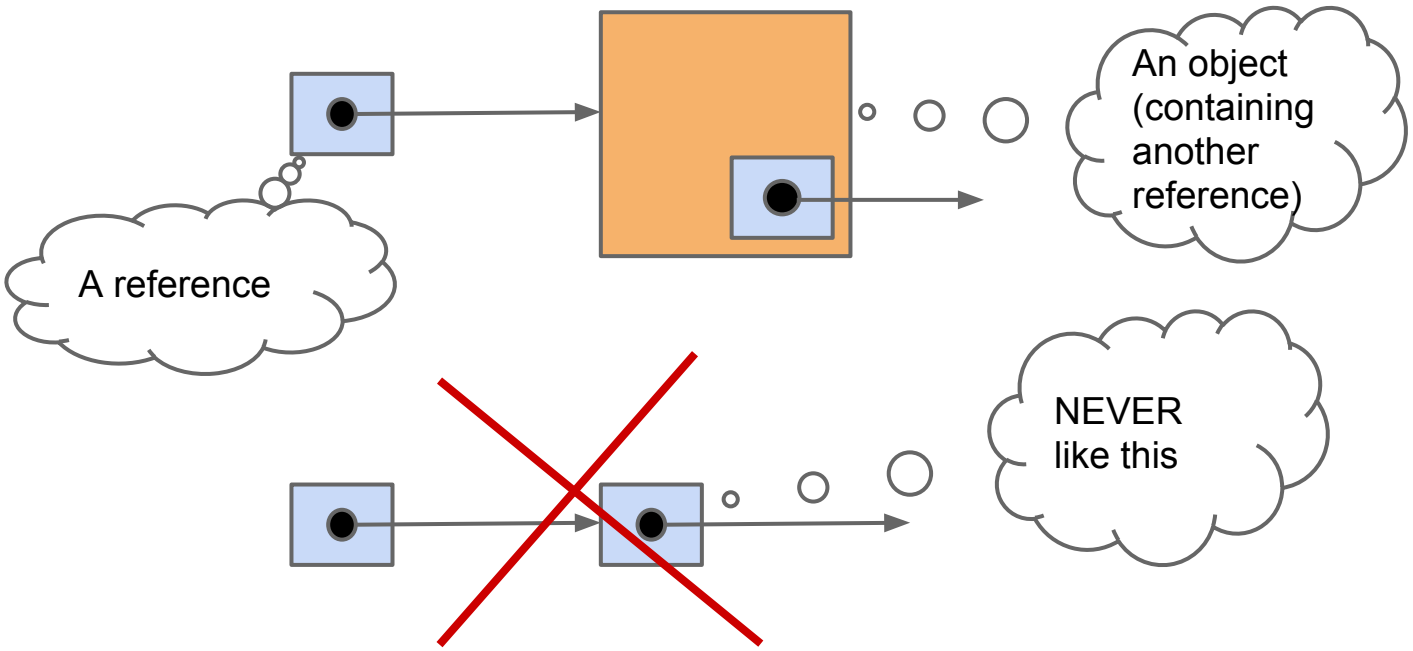


Integer j = new Integer(56);

Objects only accessible via references

References references Objects

A Java reference always references an object, never another reference*



*) Possible in languages like C, C++

Scope

"In computer programming, the scope of an identifier is the part of a computer program where the identifier, a name that refers to some entity in the program, can be used to find the referred entity. " // Wikipedia

Many different scopes in Java, most well known is

```
// Java block scope
{
…
}
```

Shadowing*

If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration **shadows** the declaration of the enclosing scope. You cannot refer to a shadowed declaration by its name alone

If many occurrences of same name Java normally uses “the closest” declaration

Imperative vs Functional Variables

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

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)

State is Problematic

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

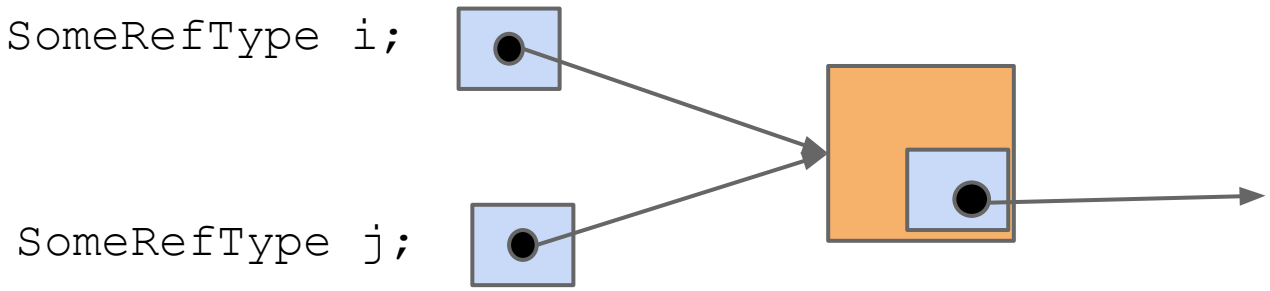
- Holding the (partial) result and information how to proceed with the calculation
- During execution the content will change
- Have to to put correct value of the correct kind in the correct variable in the correct order...

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

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

In Fact it's Even Worse*

References can reference the same object. We possible have **shared state** (the alias problem)



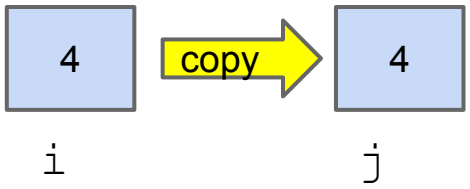

```
j.setValue( ... ); // Will affect i !!!
```

"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

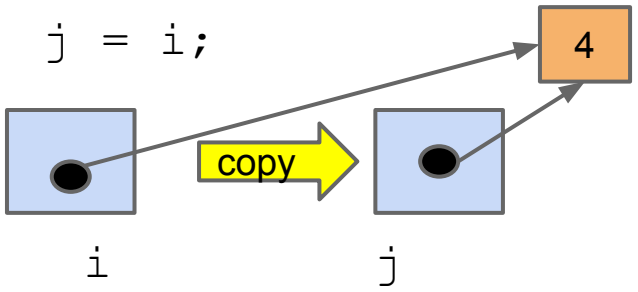
Assignments

Always : Value (content) is copied from right side variable to left side variable (location)

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



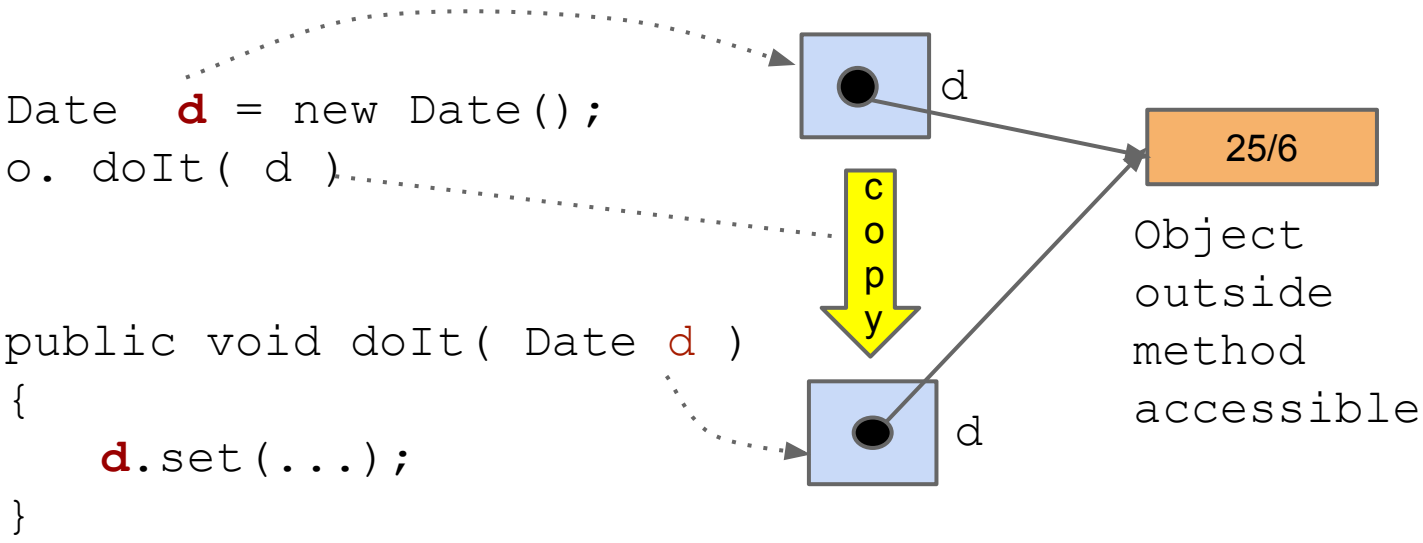

```
Integer i = 4;
Integer j;
//After this we have 2
//references
j = i;
```



A variable can act as a **left value** (a location to store something) or a **right value**, the value (content) of the variable

Call by Value

When calling methods values are also copied, i.e. **call by value**. If references copied there are implications ...



Out Parameters*

Normally method parameters should be "read only" (some put **final** on params)

- 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 (avoid outparam as return)

... ok usage

- Passing an array/Collection to be manipulated by method is ok (sorting, …)

By Value or by Reference*

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

Value semantics **(by value**), values copied/compared

- No shared state

Reference semantics (**by reference**), references copied/compared

- Shared state

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 ){

}
```

Why References?

There's a lot of copying going on!

If using primitive type variable possible many bytes to copy (assume an image as a primitive type, many MBs to pass around)

Copying a reference in Java is 32 or 64 bits (regardless of size of object). Much more efficient

Equality

A fundamental question is when two objects are equal (identical)

- By value or by reference?

Default in Java: By reference

More later...

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 = …
// Get a value (b) and modify list (state change)
boolean b = is.add(123);

// Assignment mainly used for side effect
```

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 (or avoid) change state of object, no (avoid) side effects
- Used to retrieve information (state or calculated)
- ... more later

Mutator-method (setters)

- Changes state of object
- Doesn't return information (or avoid) about object (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)

// 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;
}
```

Like an equation. How to prove correctness of this?

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

Declarative Style Proof

Prove: pow a b = a^b

By induction

- Base: pow a 0 = 1 = a^b (b = 0)
- Assume: pow a b = a^b (b > 0)

Show: pow a b+1 = a^{b+1}

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

a * pow a b = a * a^b = a^{b+1}

Invariants

*"In computer science, an **invariant** is a condition that can be relied upon to be true during execution of a program, or during some portion of it. It is a logical assertion that is held to always be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.' //* Wikipedia

Invariants tells us that something is fixed, we don't need to check it all the time ...

Imperative Style Proof*

Prove: pow(a,b) = a^b

By use of **loop invariants.** 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 <= 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 =>
i <= b && result == a^i

Assume invariant true before iteration.
result *= a; => result == $a^{(i+1)}$ (inv. false)
i++; (inv. true)
Invariant true after loop

At termination (invariant still holds): i <= b && result == a^i && !(i < b)
=> i == b && result == a^i => result == a^b

Hard Parts of Imperative Proofs

The negation of the guard (i >= b) and the invariant should imply (=>) the desired outcome (result == a^b)

- 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

Very tedious, have to prove each statement

No side effects allowed

- 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
// Accumulator parameter holding the result
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

Tail Recursion to Imperative Style

Tail recursion easy to convert to imperative loop

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

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

```
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 .... hard again

Proof vs Reasoning

Imperative proofs quickly becomes very complicated ...

... but using the techniques for informal 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
- 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 to test (class under test, CUT)
- Write another "test"-class, testing all (public) methods in CUT
  - Test class ideally has one test method for each method in CUT
  - In test methods we put assertions (boolean expressions)
- Let JUnit run the test class
  - Test passes if assertion holds
  - JUnit will report
- Should be possible to automate tests (test suites)

Organizing Test Code

Always keep test code separate from application code

- Use Test Packages folder in NetBean
- 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, JaCoCo, plugin for NetBeans (install, see README in sample code

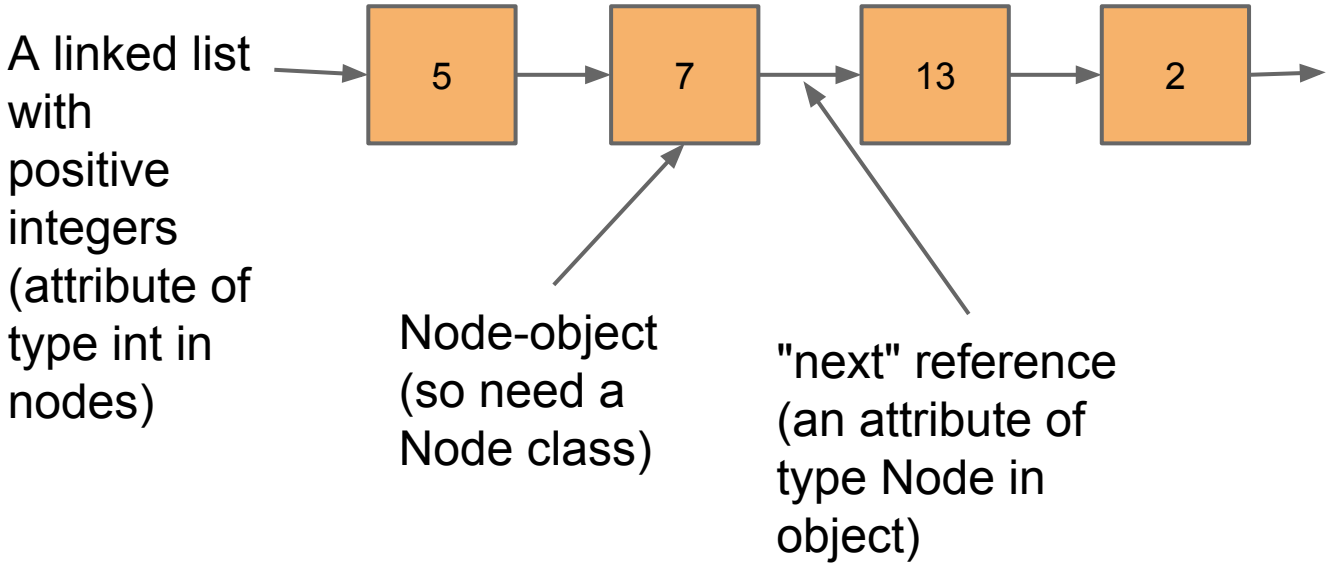
Other Benefits of Testing

- All passed tests should pass after any modification, confidence in doing changes (refactoring), just re-run the tests!
- 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)



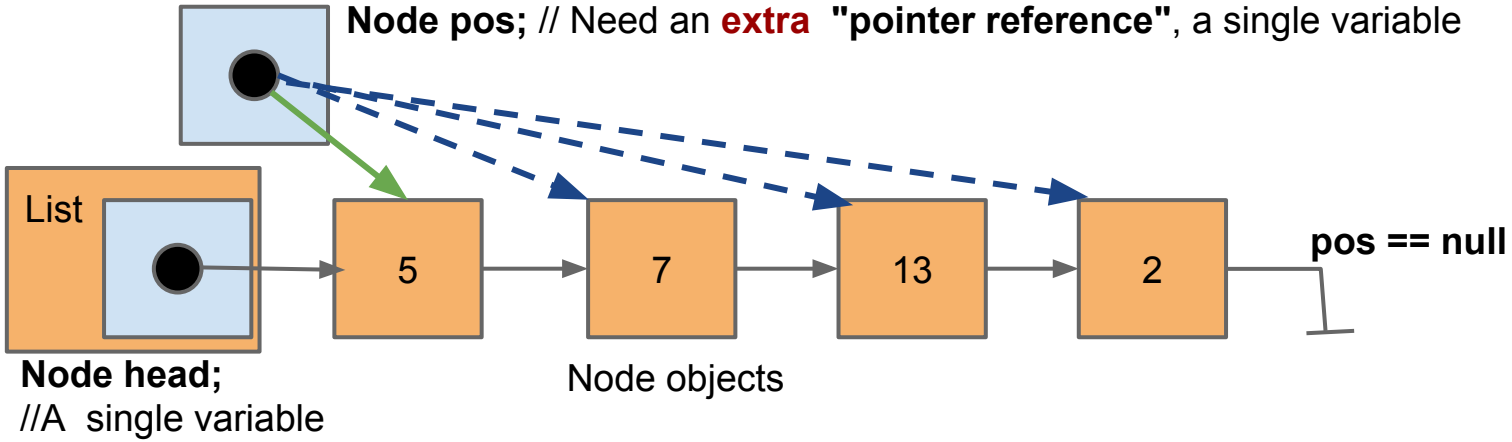
The Node Class

```
// 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(){  // Insert first
        Node n = new Node(head);
        head = n;
    }
}
```

Traversing a Linked List*

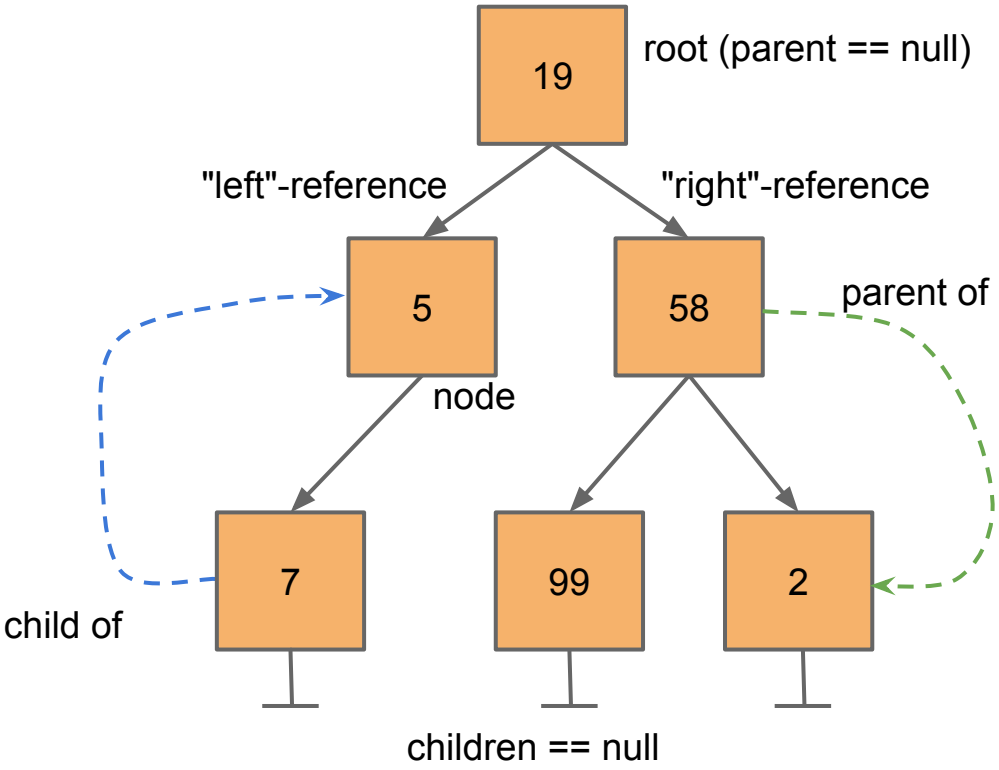


Can't change head variable, if so whole structure lost (should be final)

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

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 (also 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