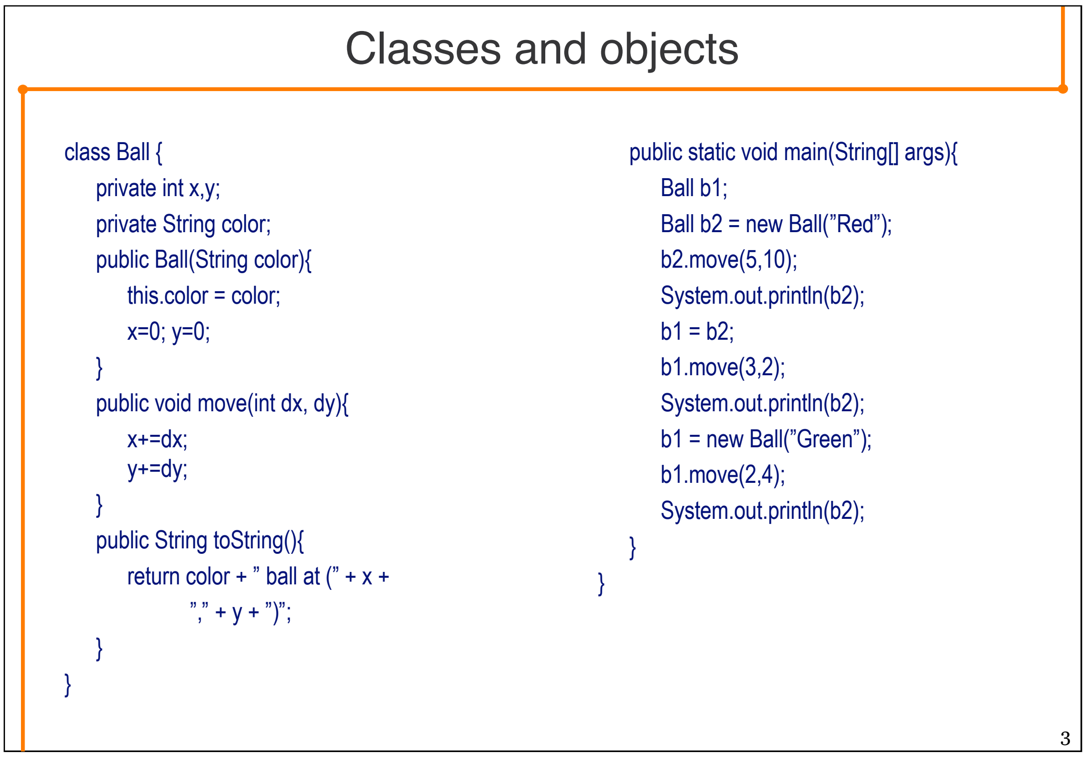
OPI
Lecture 13
Objects, Classes and References

Kasper Osterbye
Carsten Schuermann
IT University Copenhagen

1

Introduction

- Teaching staff: Carsten Schuermann, Jeffrey Sarnat
- What will be taught in the remainder of this course
  - More object-oriented languages
  - More programming
  - Some new topics, Generics IO, Threads, Design by contract and Exceptions, Reflection, Graphics, Graphical User Interfaces.
  - OOP and Java – quite some emphasis on the Java language – learn one language well.
- I will try to have room for, at each lecture:
  - Leave time for questions. Please ask in Danish if that is most comfortable.
  - New details of known stuff
  - New stuff



Classes and objects

```
class Ball {
    private int x,y;
    private String color;
    public Ball(String color){
        this.color = color;
        x=0; y=0;
    }
    public void move(int dx, dy){
        x+=dx;
        y+=dy;
    }
    public String toString(){
        return color + ” ball at (” + x +
                ”,” + y + ”)”;
    }
}
```

```
    public static void main(String[] args){
        Ball b1;
        Ball b2 = new Ball(”Red”);
        b2.move(5,10);
        System.out.println(b2);
        b1 = b2;
        b1.move(3,2);
        System.out.println(b2);
        b1 = new Ball(”Green”);
        b1.move(2,4);
        System.out.println(b2);
    }
}
```

3

This class has one part (the left) that represents a ball in a window. The ball has an x coordinate, representing its distance from the left side of the window, and a y coordinate, that represents its distance from the top of the window.

A String represents its color, as ”Red”, ”Blue” etc.

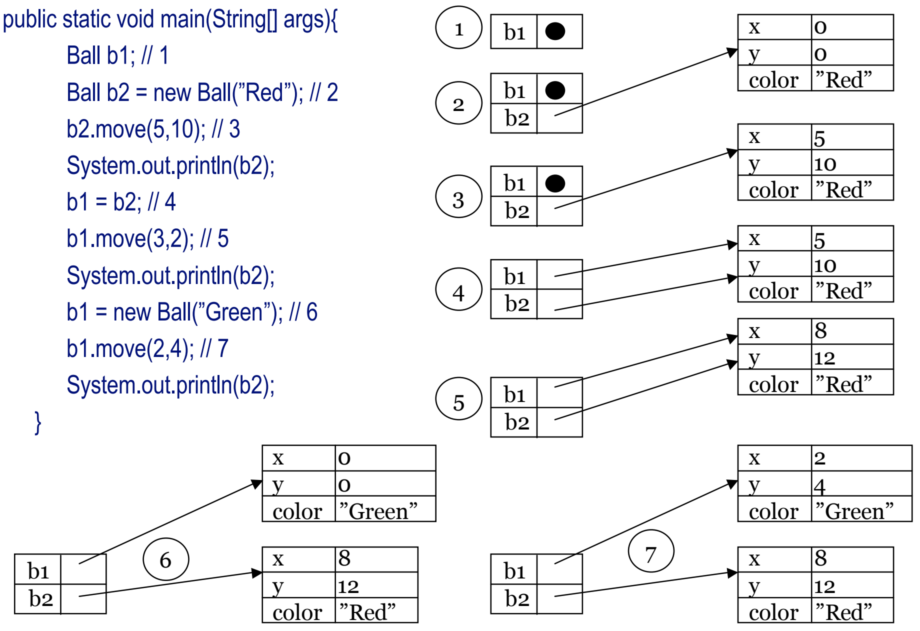
In the constructor, notice the assignment ”this.color = color”.

The move methods takes two parameters, which indicate how much should be moved. The name dx is a traditional abbriviation for delta-x, where delta in mathematics is used for a difference, ie. those who are members of the secret mathematic society will know that dx is a difference to x.

The main method declares two variables b1, and b2 to hold balls. Notice that one **do not** need to give a reference variable a value when it is declared, hence both b1 and b2 are declared correctly.

Drawing objects and references

```
public static void main(String[] args){
        Ball b1; // 1
        Ball b2 = new Ball(”Red”); // 2
        b2.move(5,10); // 3
        System.out.println(b2);
        b1 = b2; // 4
        b1.move(3,2); // 5
        System.out.println(b2);
        b1 = new Ball(”Green”); // 6
        b1.move(2,4); // 7
        System.out.println(b2);
    }
```



One of the most important things to notice in these slides are the figures themselves.

It is important to be able to draw variables, references and objects. Practicing this will enable you to get a more operational understanding of theses concepts.

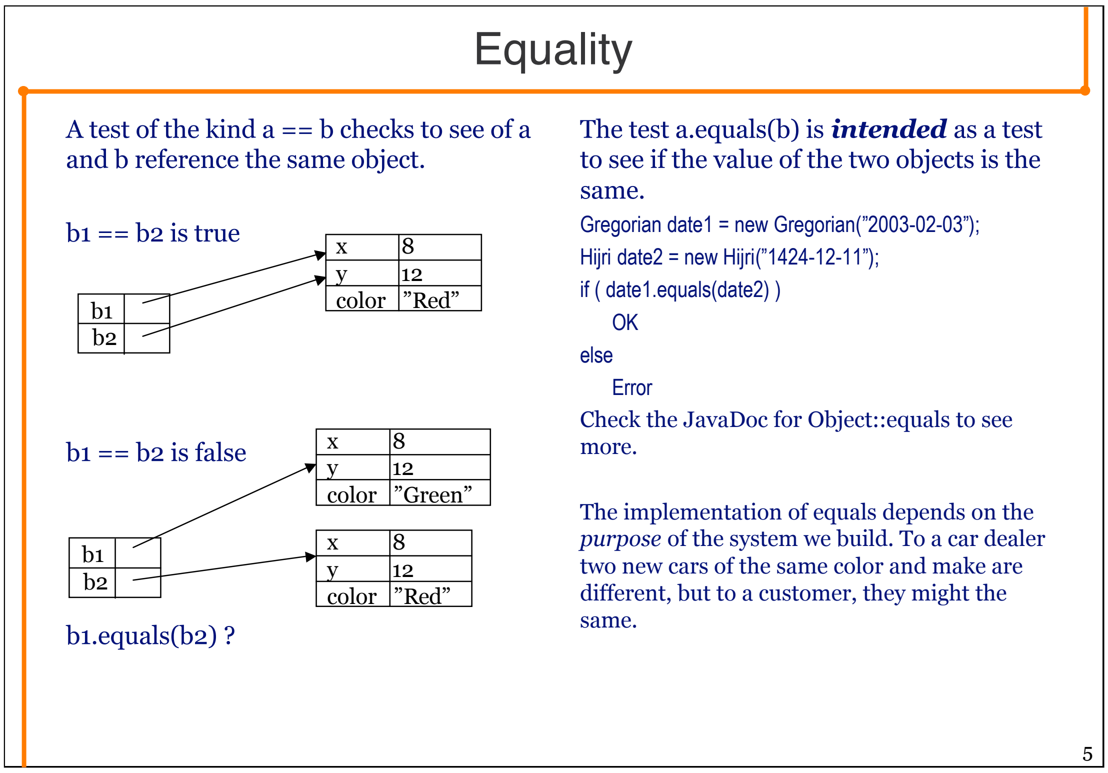
At 1, only the reference variable b1 exist. It has not any initializer, hence it has its default value, which is null – in the drawing indicated by a big dot. All reference variables have the value null if they are not initialized.

In 2, the reference variable b2 is declared, a new object of type Ball is created using the new expression, and b2 is set to refer to this new object.

In 4, b1 is set to refer to the same object that b2 refers to.

In 5, the object refered to by b1 is moved 3 units on the x axit, and 2 on the y axis. Notice the term ”the object refered to by b1”. Neither b1 nor b2 are names of objects, they are the names of reference variables. The method call b1.move(3,2) thus means calling the method move on the object that b1 refers to.

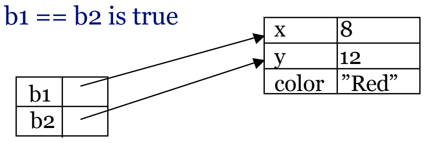
In 6 a new Ball object is created, and b1 is set to refer to this object.



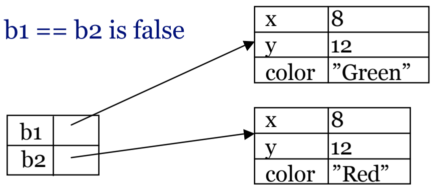
Equality

A test of the kind a == b checks to see of a and b reference the same object.

b1 == b2 is true



b1 == b2 is false



b1.equals(b2) ?

The test a.equals(b) is ***intended*** as a test to see if the value of the two objects is the same.

```
Gregorian date1 = new Gregorian(”2003-02-03”);
Hijri date2 = new Hijri(”1424-12-11”);
if ( date1.equals(date2) )
    OK
else
    Error
```

Check the JavaDoc for Object::equals to see more.

The implementation of equals depends on the *purpose* of the system we build. To a car dealer two new cars of the same color and make are different, but to a customer, they might the same.

5

The class Object implements the equals method. That means that one can use both a == b and a.equals(b) for any kind of references; with the one exception that one cannot do null.equals(null) – no method call on null references.

The implementation of equals in object is to use ==.

The following rules regarding the equals method is taken from JavaDoc

”The equals method implements an equivalence relation:

1.It is *reflexive*: for any reference value x, x.equals(x) should return true.

2.It is *symmetric*: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

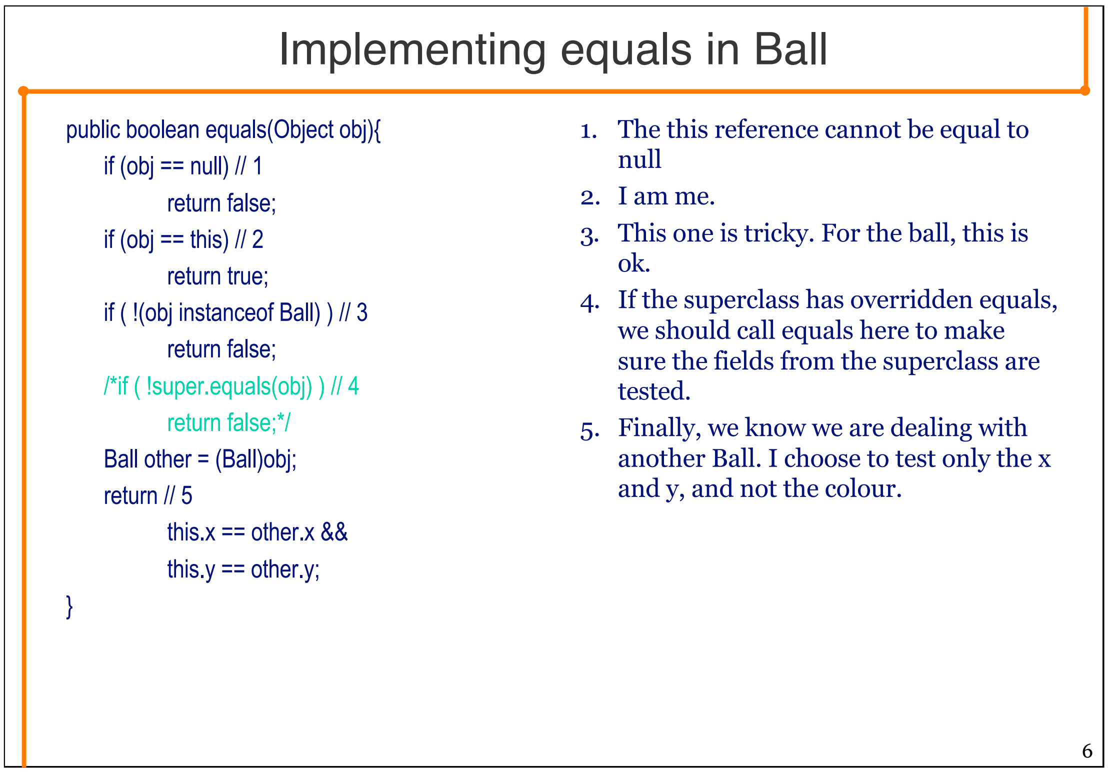
3.It is *transitive*: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

4.It is *consistent*: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.

5.For any non-null reference value x, x.equals(null) should return false.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes. ”

The last comment regarding hashCode should be ignored for the moment, we will return to that later in the course.



Implementing equals in Ball

```
public boolean equals(Object obj){
    if (obj == null) // 1
        return false;
    if (obj == this) // 2
        return true;
    if ( !(obj instanceof Ball) ) // 3
        return false;
    /*if ( !super.equals(obj) ) // 4
        return false;*/
    Ball other = (Ball)obj;
    return // 5
        this.x == other.x &&
        this.y == other.y;
}
```

1. The this reference cannot be equal to null
2. I am me.
3. This one is tricky. For the ball, this is ok.
4. If the superclass has overridden equals, we should call equals here to make sure the fields from the superclass are tested.
5. Finally, we know we are dealing with another Ball. I choose to test only the x and y, and not the colour.

6

There are some notes on this implementation of equals.

First, there is a predefined test in the equals method of class Object. It will test both 1 and 2, but we cannot really reuse this. If the equals in Object::equals return true, we know it is because the two objects are the same. But if it returns false, we cannot know if it is because obj is null or just not the same object as this. Conclusion, we cannot write a shorter test by reusing the Object::equals method.

The order in which these tests take place is designed to use the fastest test first, and the slow ones later. Testting using == is very fast, thus 1 and 2 are done first.

I will diskuss 3 below.

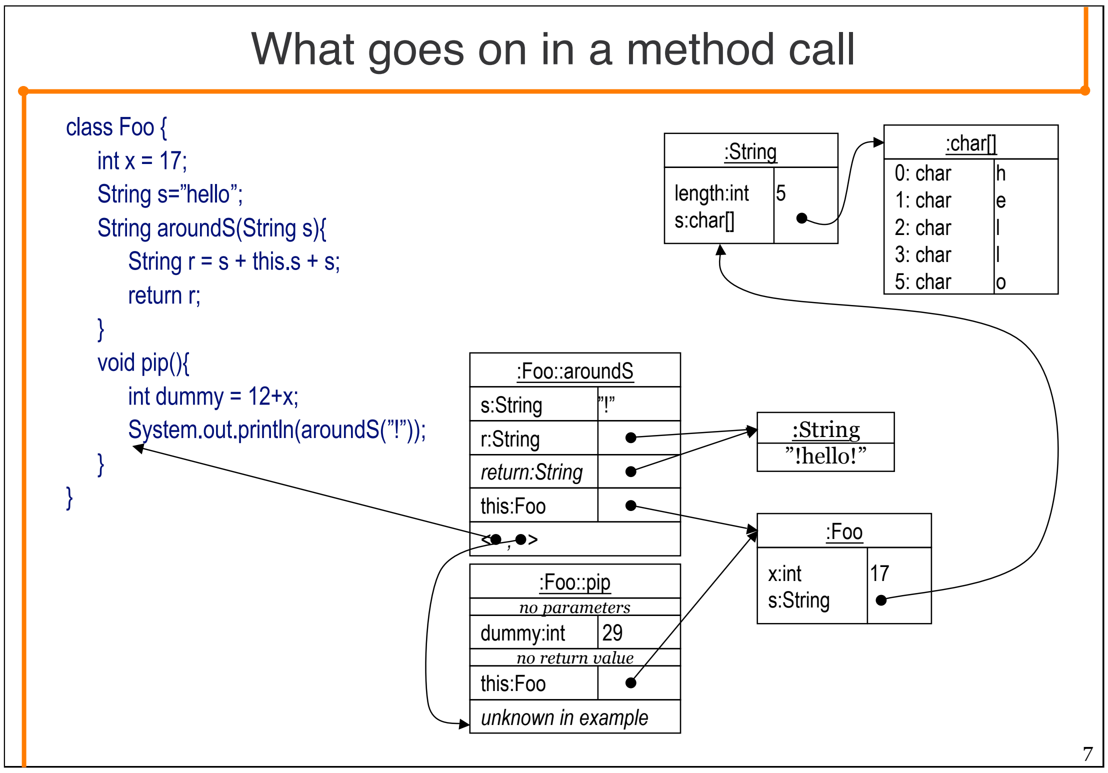
Test number four is necessary when we inherit from a class X, which is not Object, because X might have private fields which we cannot test here. When, as in the Ball example, we inherrit directly from Object, we *must not* do it, because the equals in Object is the same as ==.

When we reach 5, we know that the other object is a Ball (because of test 3). Thus we can cast obj to a Ball object. We return true if the x and y is the same, false otherwise.

As I said on the previous slide, the equals method is ment to allow you as a programmer the choice to define what should be equals in your program. In my ball example here, the color does not matter, hence I do not check on that.

Regarding test 3, then this test checks to see if the other object is an instance of Ball, *or any subclass of Ball*. This is my choice. Sometimes I will not allow objects of subclasses to be equal. In that case I can use a test (this.getClass() == obj.getClass()). This is about 50% faster.

But in the case of the calendars, I might want to allow dates from different Calender types to be equal. In that case, testing become quite complicated, and with the Islamic Hijri calender, we need to do lunar calendar computations, which are not at all simple. And also the Gregorian calendar with its leap-year computations is complicated.



What goes on in a method call

```
class Foo {
    int x = 17;
    String s="hello";
    String aroundS(String s){
        String r = s + this.s + s;
        return r;
    }
    void pip(){
        int dummy = 12+x;
        System.out.println(aroundS("!"));
    }
}
```

Just like we can depict objects as boxes with variables in it, we can depict ***method invocations***. Where we for an object draws each instance variable, there are five things of interest with methods: 1) Parameters; 2) Local variables; 3) return value; 4) The "this" reference; 5) return information.

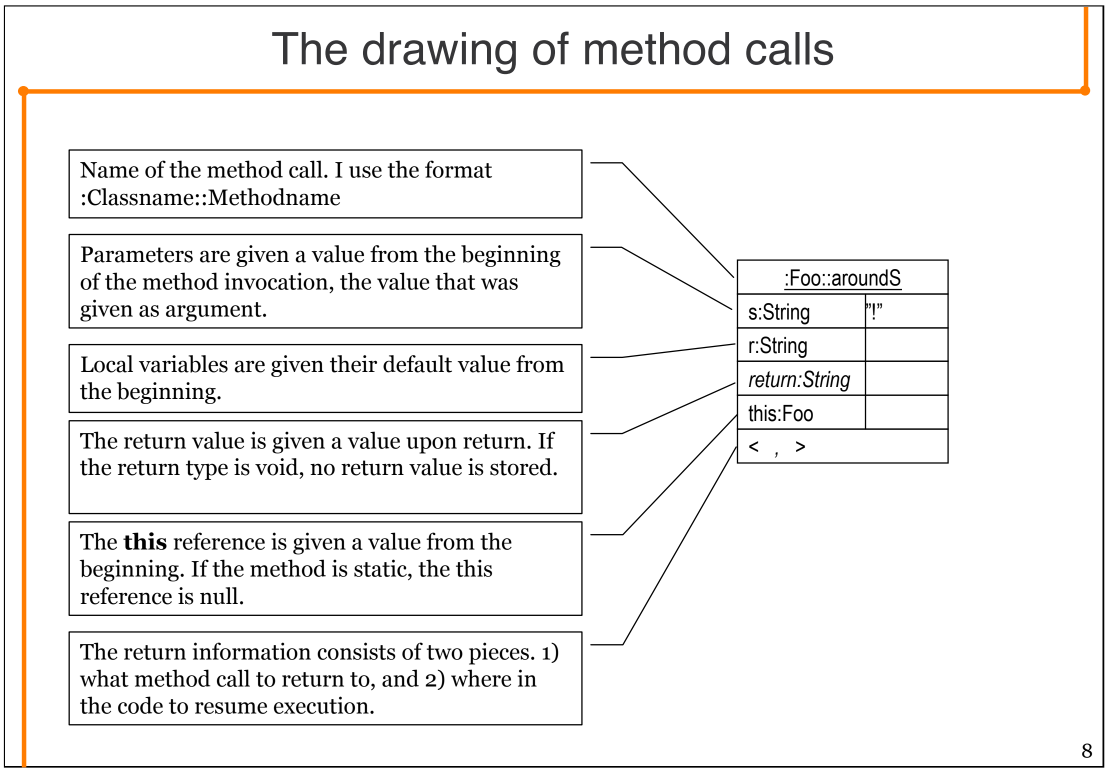
In the example, some method called the pip method on an object of kind Foo. The pip method has one local variable "dummy", which has been given the value 12+x. x is the instance variable, which is really this.x ~ having the value 17. Notice the this reference; "this.x" means to find the x variable in the object referenced by the this reference.

I have drawn strings in several different ways. The Foo object (named :Foo) has the instance variable s, which is a *reference* to a String object. The String object in turn has a *reference* to a char array. It is quite cumbersome to draw this way, so I will use the notation from the aroundS method invocation, either directly giving the string, or when I want to emphasize that two variables points to the same String object, I draw it as can be seen in the return value and r variable.

When the aroundS method call is finished, the program execution must be resumed where the method was called. The return information consists of two parts, a) the method invocation which we must return to (in our case pip), and b) the location in the code where we will resume executing (in our case the println call).

The this reference can be accessed in Java using the keyword this. One cannot assign a new value to *this* in Java (or any other OO programming language for that manner).

Method invocations are often drawn on top of each other, with the topmost representing the method invocation which is currently active. Because of this, method invocations are sometimes called *stack frames*, and alle the method invocations is named a *call stack.*



The drawing of method calls

Name of the method call. I use the format :Classname::Methodname

Parameters are given a value from the beginning of the method invocation, the value that was given as argument.

Local variables are given their default value from the beginning.

The return value is given a value upon return. If the return type is void, no return value is stored.

The **this** reference is given a value from the beginning. If the method is static, the this reference is null.

The return information consists of two pieces. 1) what method call to return to, and 2) where in the code to resume execution.

8

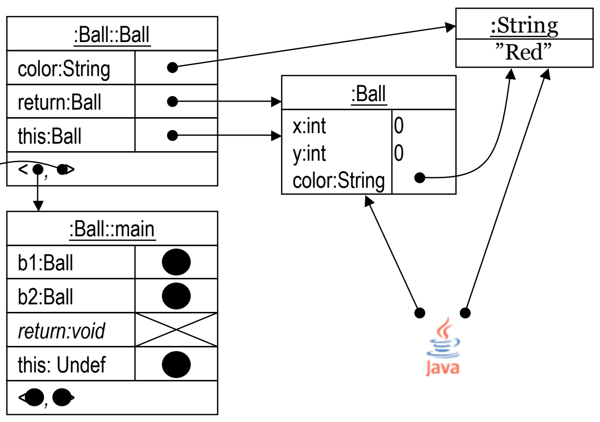
The notation used in the pictures is UML inspired. In UML, an object is drawn as a box with the name and class in the top box, and instance variables (fields) in the box below. In UML one can give a name to an object, as a:Foo (the underlining is UML syntax, and means that it is an object, not a class). In Java, as in all other object oriented programming languages, one cannot name an object, so we omit the name, and just give the class :Foo. The notion for methods invocation, :Class::method. The underline indicates that a method invocation

Drawing methods invocations (or calls – two words for the same thing) is very time consuming. But it is a necessary exercise to get a detailed understanding of how method calls actually work. Also, it is a very good operational understanding when we in a few slides time get to the concept of recursion.

Method calls

```
class Ball {
    private int x,y;
    private String color;
    public Ball(String color){
        this.color = color;
        x=0; y=0;
    }
    public void move(int dx, dy){
        x+=dx;
        y+=dy;
    }
    …
}
public static void main(String[] args){
    Ball b1;
    Ball b2 = new Ball(”Red”);
    b2.move(5,10);´
…}
```

Follow the first two statements in main



9

Notice that the this reference is null in the main method. In a static method the this pointer is undefined. The compiler is able to check this at compile time, and gives an error message if one tries to refer to fields or methods in the class which need the this reference.

In connection with the constructor call, the following steps takes place.

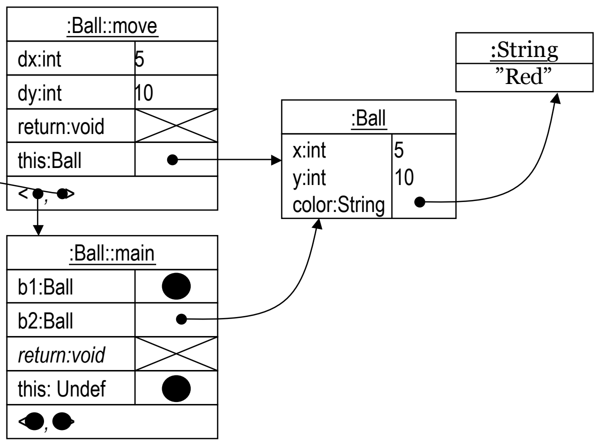
1. The compiler will have instructed the virtual machine that it needs a String with the value ”Red”. This object therefore exists beforehand.
2. The virtual machine makes a new Ball object, and initializes the fields according to the initializers (in the Ball class there are none).
3. The constructor is called. Above this is depicted as a method call.
4. When the constructed terminates, the **this** referece is the return value, to be assigned to b2.

Notice the three steps that makes the color of the object Red. First, a String with ”Red” is created. Second, a reference to the string is passed as argument to the constructor. Finally, the field color is set to refer to the String object.

Method calls

```
class Ball {
    private int x,y;
    private String color;
    public Ball(String color){
        this.color = color;
        x=0; y=0;
    }
    public void move(int dx, dy){
        x+=dx;
        y+=dy;
    }
    …
}
public static void main(String[] args){
    Ball b1;
    Ball b2 = new Ball(”Red”);
    b2.move(5,10);´
…}
```

Follow the third statement in main



10

Notice, the this reference is null in the main method. In a static method the this pointer is undefined. The compiler is able to check this at compile time, and gives an error message if one tries to refer to fields or methods in the class which need the this reference.

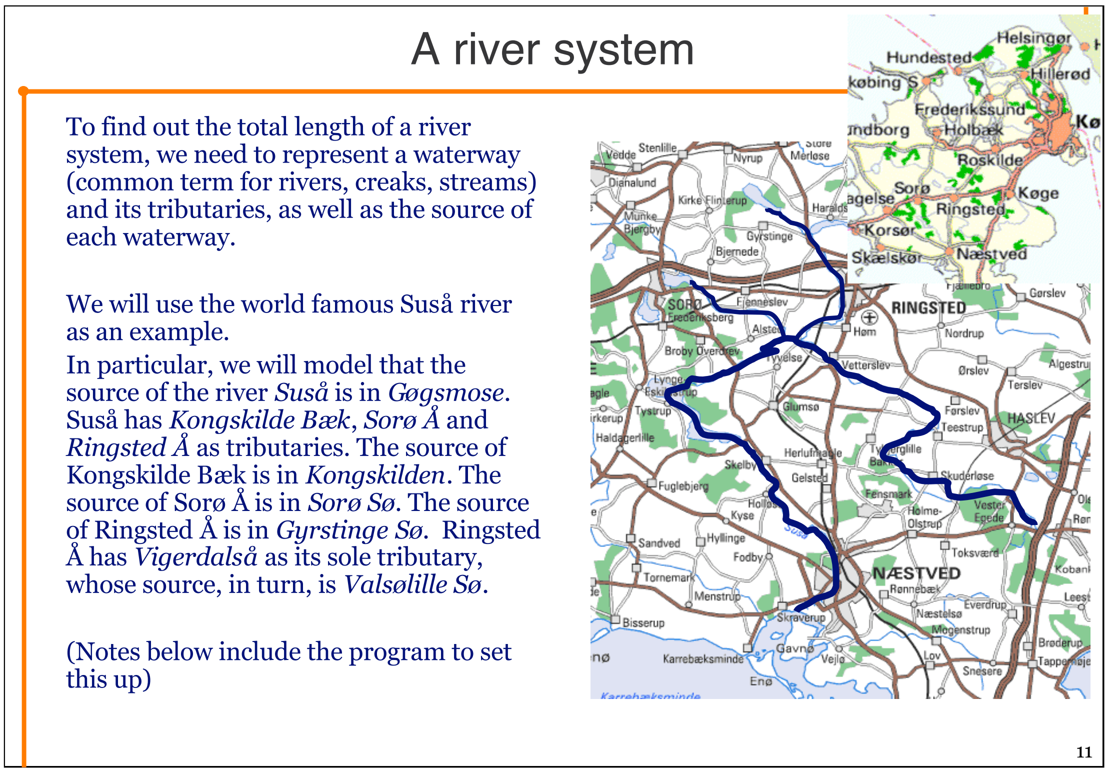
In a method call o.foo(...), the **this** reference will be the same as the o reference.

In exercise 1.f I use the terms explicit and implicit this references.

In the move method, we make use of two *implicit* this references, namely x and y. x and y are names of fields in the class. We could have written the body with *explicit* this references as:

```
this.x += dx;
this.y += dy;
```

In the constructor, the parameter color shadows the field named color. We therefore have no other option than using the *explict* this reference to refer to the color field.



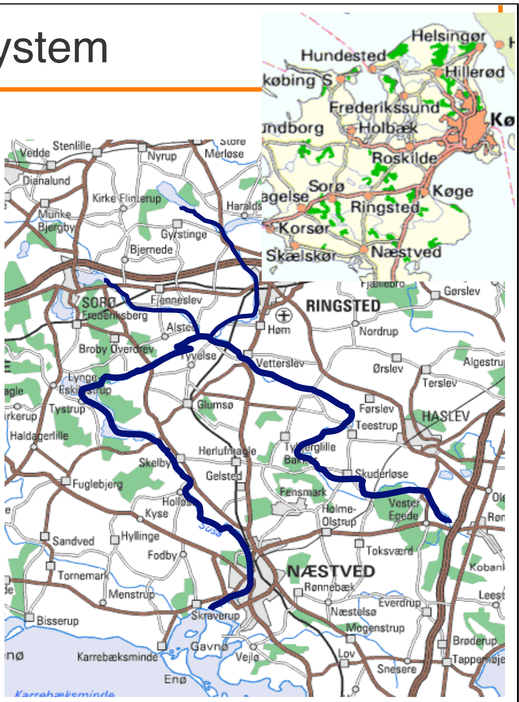
A river system

To find out the total length of a river system, we need to represent a waterway (common term for rivers, creaks, streams) and its tributaries, as well as the source of each waterway.

We will use the world famous Suså river as an example.

In particular, we will model that the source of the river *Suså* is in *Gøgsmose*. Suså has *Kongskilde Bæk*, *Sorø Å* and *Ringsted Å* as tributaries. The source of Kongskilde Bæk is in *Kongskilden*. The source of Sorø Å is in *Sorø Sø*. The source of Ringsted Å is in *Gyrstinge Sø*. Ringsted Å has *Vigerdalså* as its sole tributary, whose source, in turn, is *Valsølille Sø*.

(Notes below include the program to set this up)



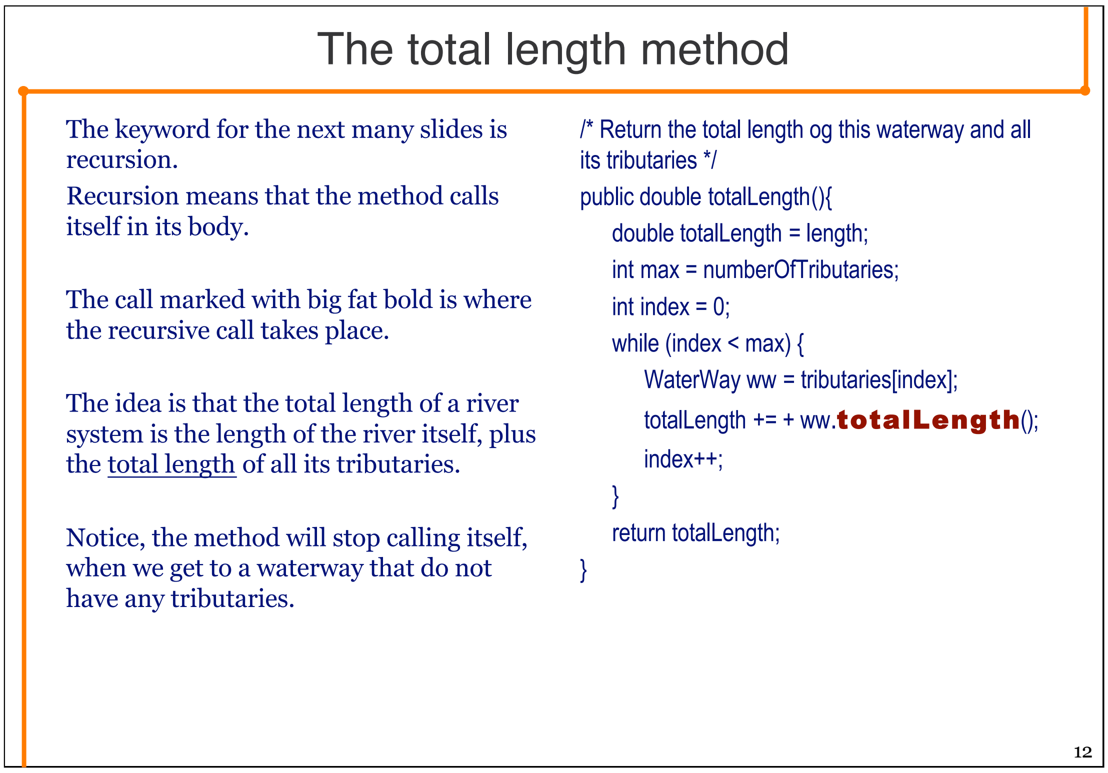
11

The reason to examine this example is to look at recursion. River systems and family trees are exellent examples for natural recursion.

```
package dk.itu.oop.lecture1;
class WaterWayTest {
    public static void main(String[] args){
        WaterWay suså = new WaterWay("Suså", 65.4, new WaterSource("Gøgsmose", 1200) );
        WaterWay kkbæk =  new WaterWay("Kongskilde bæk", 5.3, new WaterSource("Kongskilde", 680) );
        WaterWay sorøå = new WaterWay("Sorø å", 18.3, new WaterSource("Sorø sø", 2100) );
        WaterWay ringå = new WaterWay("Ringsted å", 22.6, new WaterSource("Gyrstinge Sø", 3500) );
        WaterWay vigerdalså = new WaterWay("Vigerdalså", 7.8, new WaterSource("Valsølille sø", 350) );
        suså.addTributary(kkbæk);
        suså.addTributary(sorøå);
        suså.addTributary(ringå);
        ringå.addTributary(vigerdalså);

        System.out.println("Total length of suså riversystem is: "
            + suså.totalLength() + " km."); // 119.4

        System.out.println("Total water flow of suså riversystem is: "
            + WaterWay.totalLitersPrHour(suså) + " l/h"); // 7830
    }
}
```



The total length method

The keyword for the next many slides is recursion.

Recursion means that the method calls itself in its body.

The call marked with big fat bold is where the recursive call takes place.

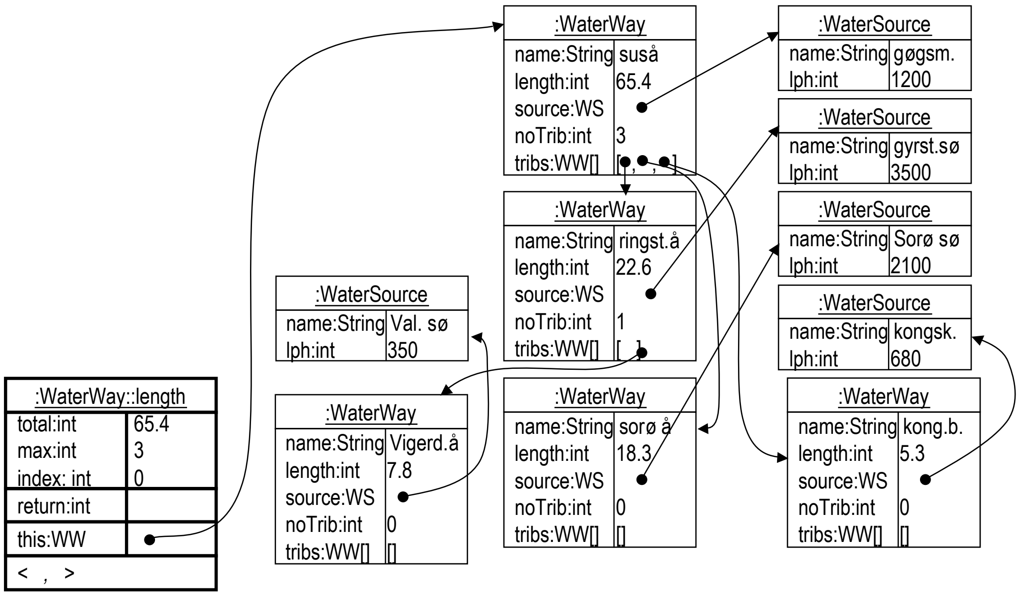
The idea is that the total length of a river system is the length of the river itself, plus the total length of all its tributaries.

Notice, the method will stop calling itself, when we get to a waterway that do not have any tributaries.

```
/* Return the total length og this waterway and all
its tributaries */
public double totalLength(){
    double totalLength = length;
    int max = numberOfTributaries;
    int index = 0;
    while (index < max) {
        WaterWay ww = tributaries[index];
        totalLength += + ww.totalLength();
        index++;
    }
    return totalLength;
}
```

12

Call of suså.length()



13

In the next many slides this object structure remains the same. It represents our simplified Suså River system.

In order to fit things on a slide, I have made a number of abbreviations on drawing. For WaterWays, the source field is of type WaterSource, here written as WS. The numberOfTributaries field is abbreviated noTrib, and tributaries are abbreviated tribs. In the WaterSource objects, lph is short for litersPerHour.

I have made the simplification that I have drawn the array inside the waterway object. An array is really an object by itself.

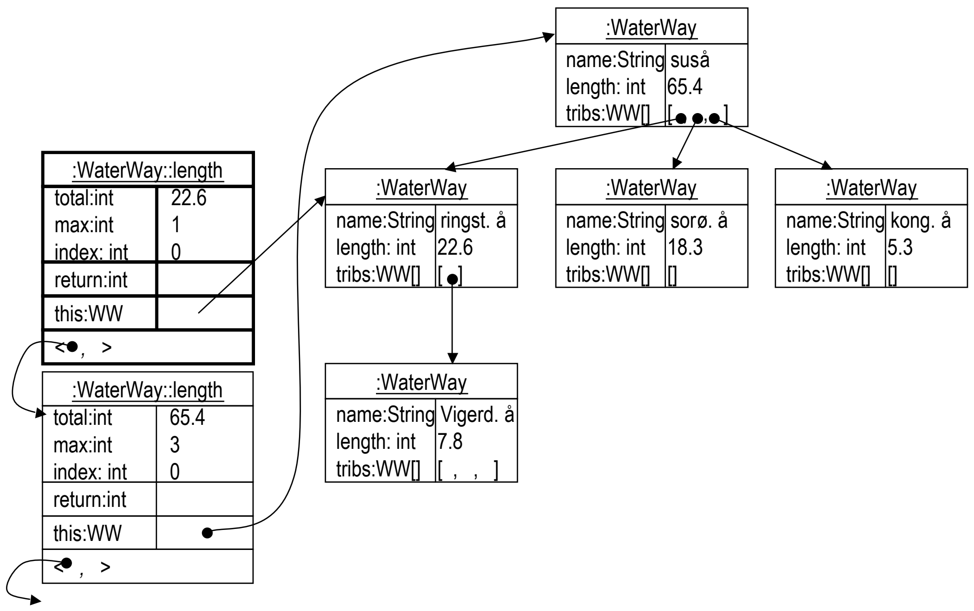
In the drawings of the method invocations I have omitted the local variable ww. This is in order to have room enough on the slides.

Also, I have abbreviated the names of both water ways and water source.

The first call to length is on the object representing Suså.

I have drawn the situation as it is just before the recursive call takes place.

recursive call 1



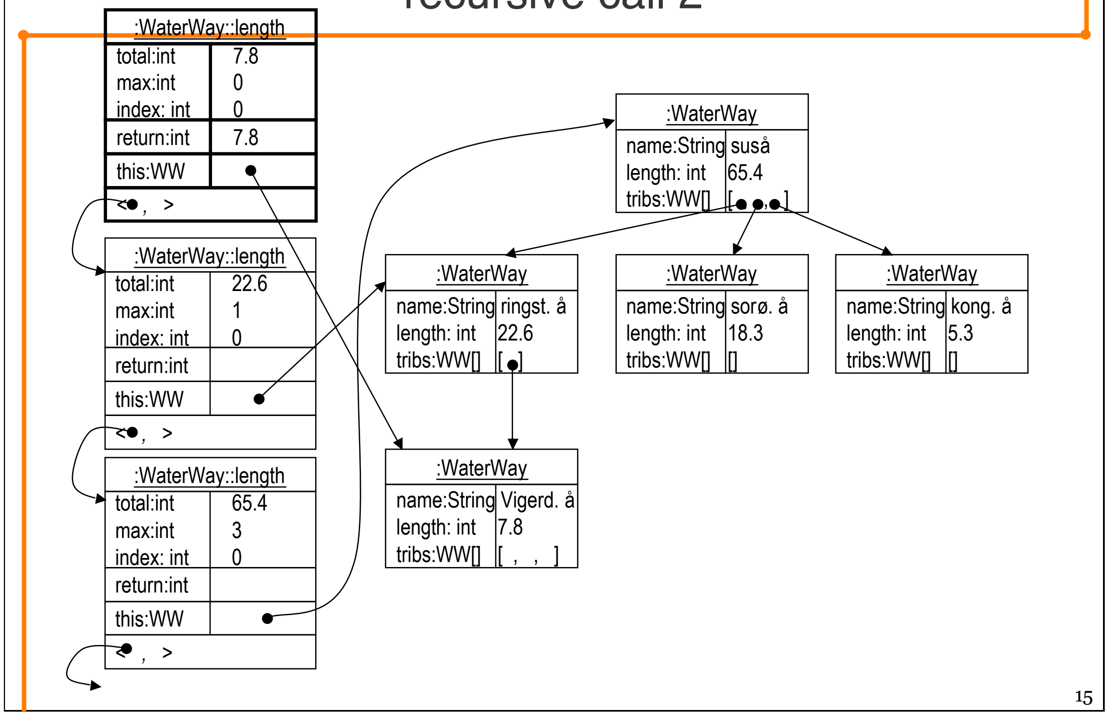
14

This drawing depicts the situation after the first recursive call of length. The call is on the first tributary to Suså, Ringsted å.

The bolded method call box is the active method call.

In this, and many other such drawings, it is normal that one does not include the information regarding where in the code one should return to. Obviously this information is necessary when the system actually runs, but there was no room left on the slide to show it.

recursive call 2

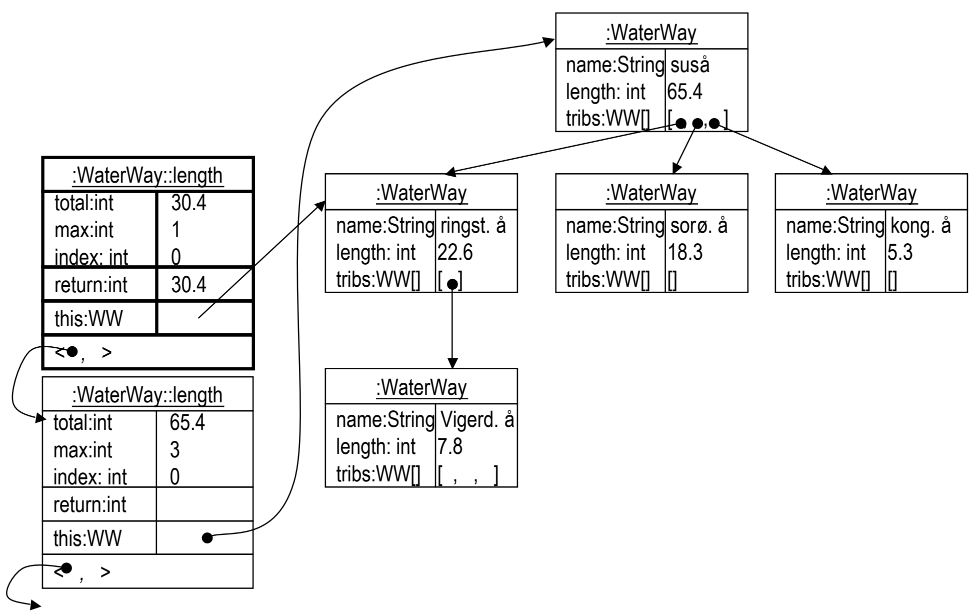


15

Ringsted å has a tributary as well; therefore there is one more recursive call of the length method.

The bolded method call is sometimes called to ”top of the call stack”. A stack is a data-structure, where one adds and remove elements from the same end. Think of a stack of dishes. Here one can add to the top, and remove from the top. In the computer, whenever we call a method, a new method invocation object is created, and is placed on the top of the stack. The first element in the return information points back to the second to the top method call. We can not make the same physical arrangement as with the dishes, and need an explicit reference.

return to previous call



16

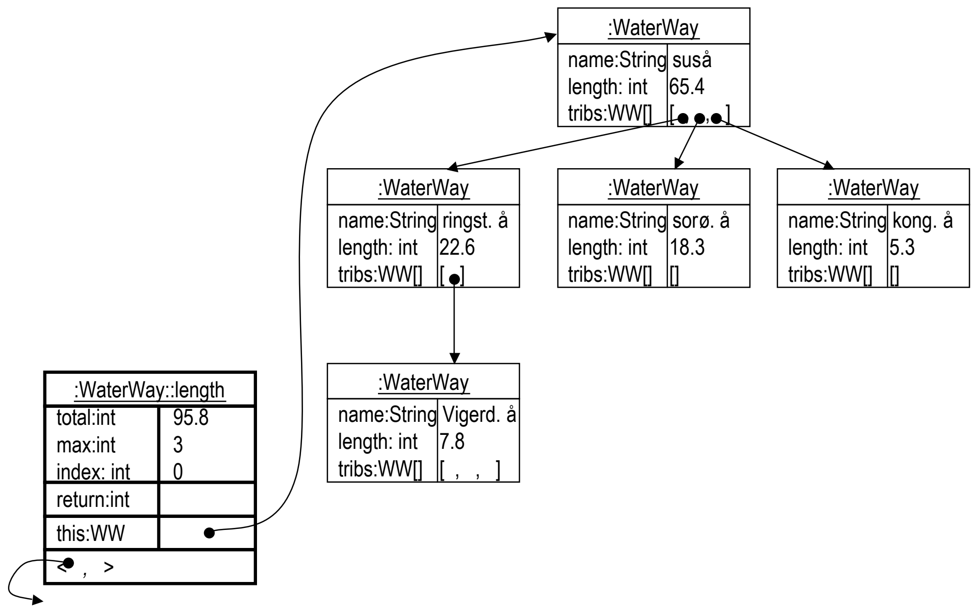
In the drawing above, we have added the return value from the previous call to the total in the topmost method invocation.

We have incremented the index counter, and it is now equal the max, and we have left the for loop, and set the return value to the same as total (an abbriviation for totalLength). We are now ready to return to where we were called from.

```
public double totalLength(){
    double totalLength = length;
    int max = numberOfTributaries;
    int index = 0;
    while (index < max) {
        WaterWay ww = tributaries[index];
        totalLength += + ww.totalLength();
        index++;
    }
    return totalLength;
}
```

The really important aspect to grasp regarding recursion is that there is indeed one method invocation box per call. Each box holds its own copy of variables, this reference, and each will remember where itself was calleded from. **Remember: one box per call, one shared piece of code.**

return to previous call

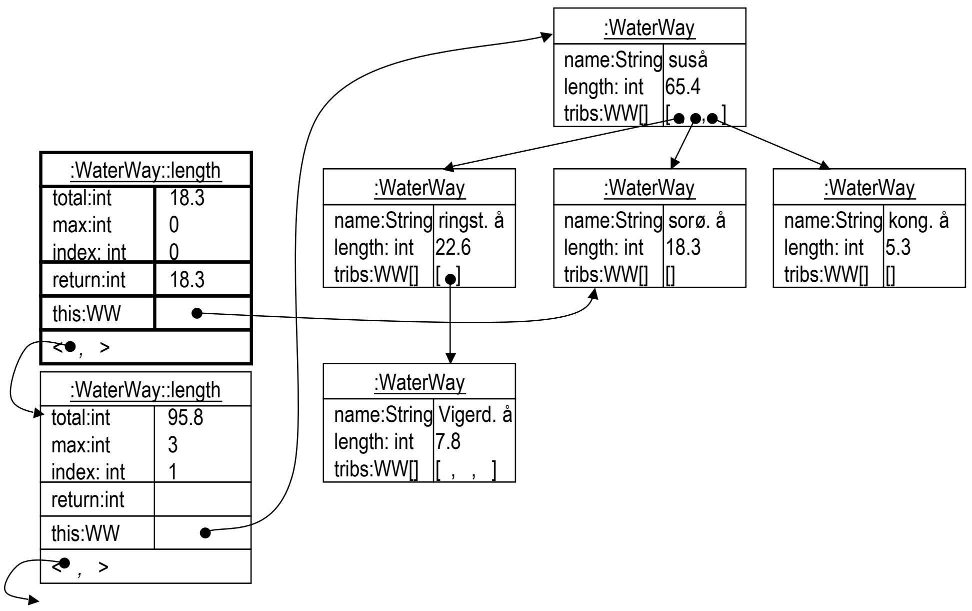


17

We have now return the the original call. Pay attention to the fact that we have incremented the index variable to 1. It is still less than max, hence we are about to call length on the second tributary.

Notice also that the return value from the previous call has been added to total.

Yet a recursive call

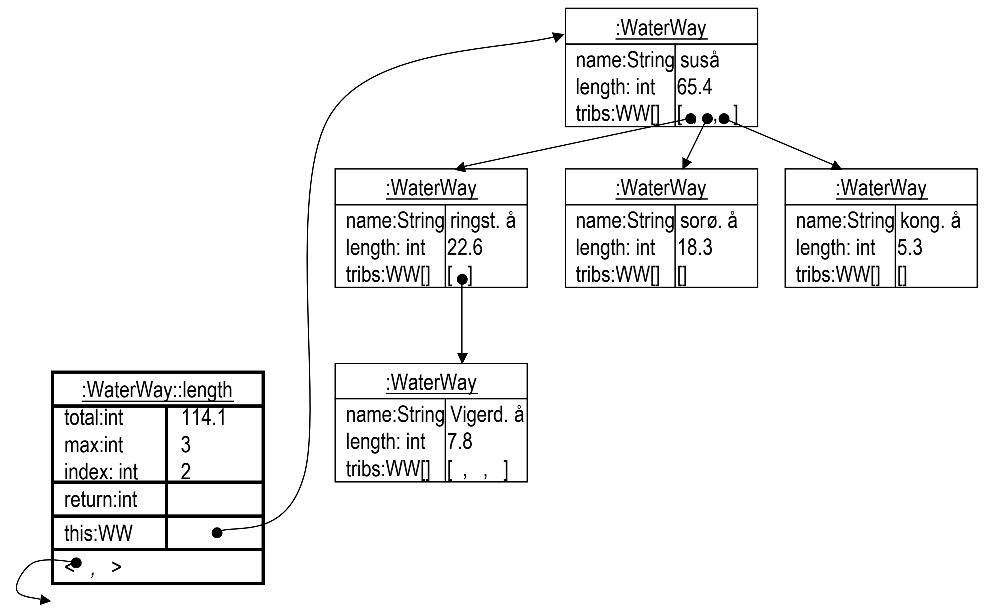


18

We have now called length on Susås second tributary. We have, as before, set the total variable to the length of this.length. Notice how the this reference is different for each call.

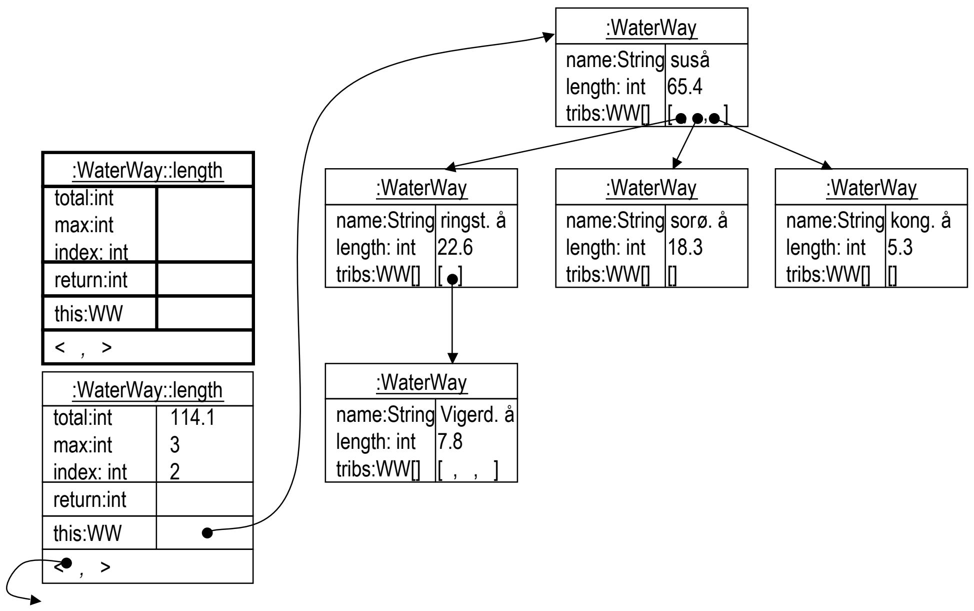
The whole idea of the length method is to let the this reference refer to all the waterways in the system. That enables us to sum the lengths from each waterway object.

and yet a return

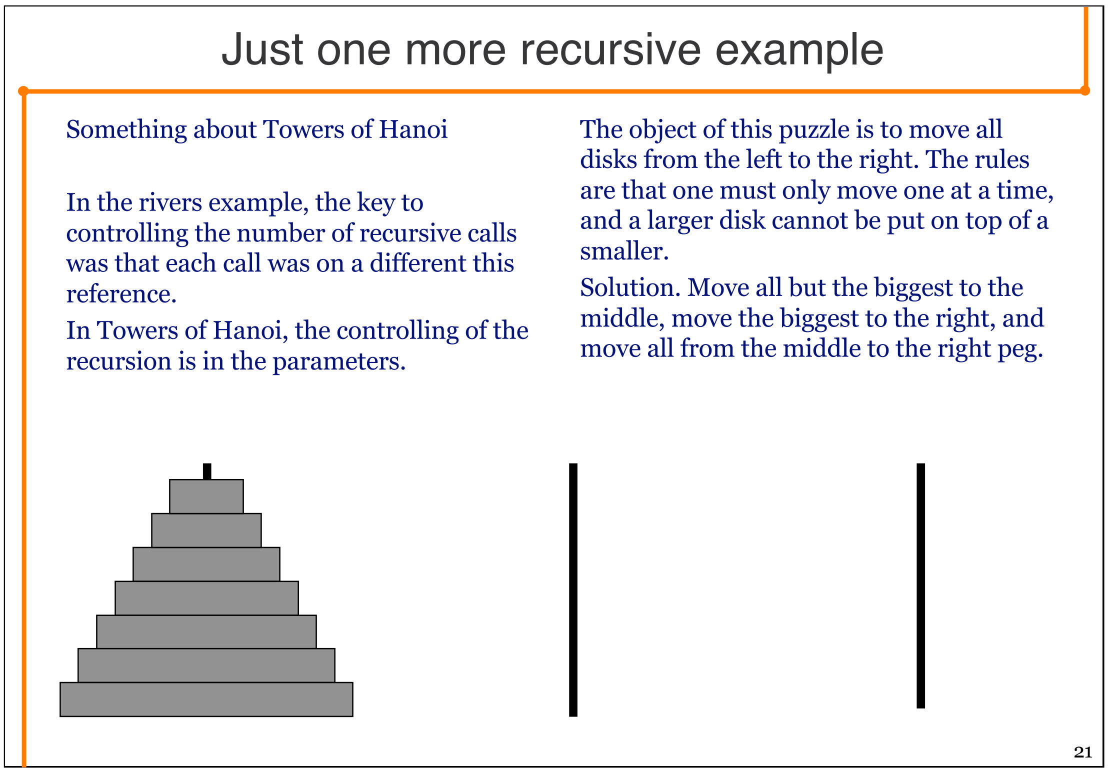


We have returned from an other call.

Exercise – fill in the last call



Your task is to figure out what the values and references are for the final call of length. The drawing should depict the situation just before we return from the call.



Just one more recursive example

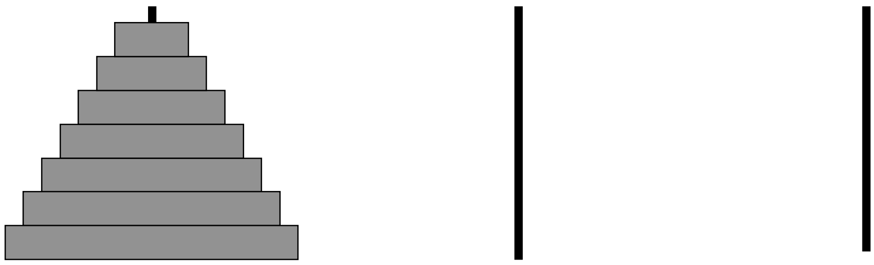
Something about Towers of Hanoi

In the rivers example, the key to controlling the number of recursive calls was that each call was on a different this reference.

In Towers of Hanoi, the controlling of the recursion is in the parameters.

The object of this puzzle is to move all disks from the left to the right. The rules are that one must only move one at a time, and a larger disk cannot be put on top of a smaller.

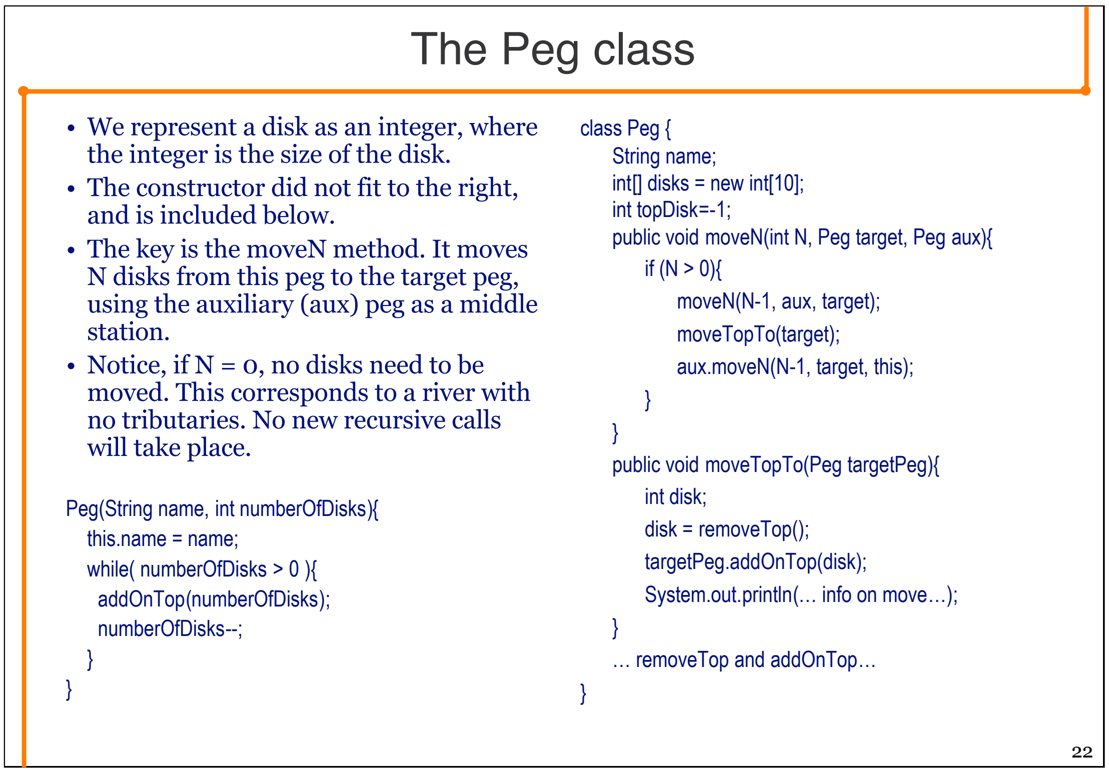
Solution. Move all but the biggest to the middle, move the biggest to the right, and move all from the middle to the right peg.



21

This is a real classic recursion problem.

Once you have grasped the the details of why the solution works, which I will attempt to explain on the next slides, you have reason to be proud. Go see a movie, get a good cup of coffee, have a cookie. You are now a real computer scientists who understands recursion. The rest is practice.



The Peg class

- We represent a disk as an integer, where the integer is the size of the disk.
- The constructor did not fit to the right, and is included below.
- The key is the moveN method. It moves N disks from this peg to the target peg, using the auxiliary (aux) peg as a middle station.
- Notice, if N = 0, no disks need to be moved. This corresponds to a river with no tributaries. No new recursive calls will take place.

```
Peg(String name, int numberOfDisks){
  this.name = name;
  while( numberOfDisks > 0 ){
   addOnTop(numberOfDisks);
   numberOfDisks--;
  }
}
```

```
class Peg {
    String name;
    int[] disks = new int[10];
    int topDisk=-1;
    public void moveN(int N, Peg target, Peg aux){
        if (N > 0){
            moveN(N-1, aux, target);
            moveTopTo(target);
            aux.moveN(N-1, target, this);
        }
    }
    public void moveTopTo(Peg targetPeg){
        int disk;
        disk = removeTop();
        targetPeg.addOnTop(disk);
        System.out.println(… info on move…);
    }
    … removeTop and addOnTop…
}
```

22

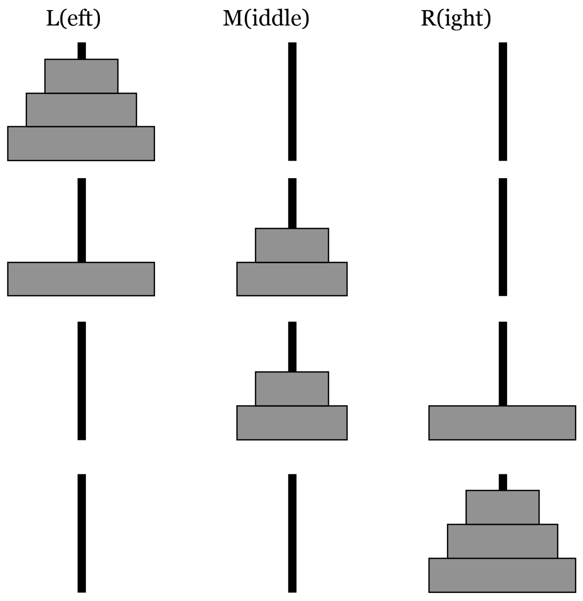
The is an important thing to grasp here. One must in the beginning learn not to worry about how the method solves to job of one less.

Convince yourself that it works properly for N=0.

Have faith, trust your Java. Assume it works for N-1.

Convince yourself that it works properly for N.

The moveN method

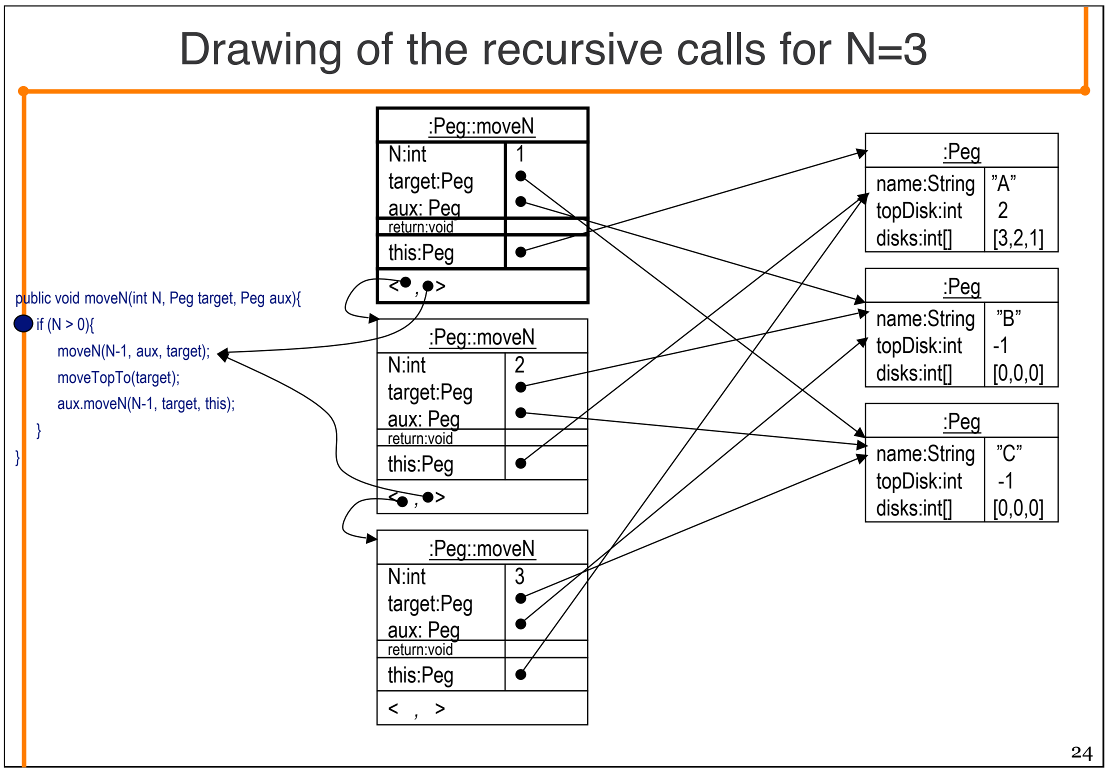


L.moveN(3-1, M, R);

L.moveTop(R)

M.moveN(3-1, R, L)

23



Drawing of the recursive calls for N=3

```
public void moveN(int N, Peg target, Peg aux){
  if (N > 0){
      moveN(N-1, aux, target);
      moveTopTo(target);
      aux.moveN(N-1, target, this);
  }
}
```

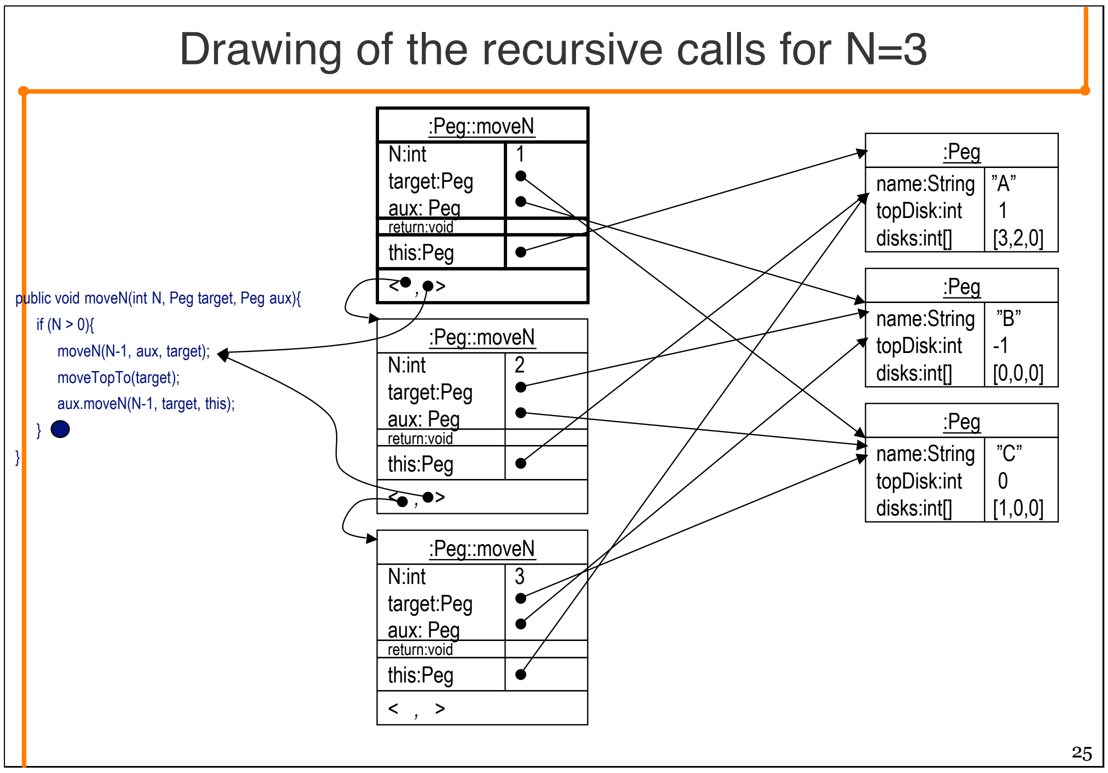
24

So far we have done nothing, Just made recursive calls.

The drawing indicates the situation just after we have entered the method, before we have done anything. This indicated by the circle in the code.

The topmost (with respect the call stack, on the drawing it is at the bottom) call is with N=1, and further recursive calls will have no effect as N will be zero, in which case the method does nothing.

Notice how the target and aux variables swap as we call. For N=3, target is the C-Peg, for N=2, target is the B-Peg, and for N=1, C is again target.



Drawing of the recursive calls for N=3

```
public void moveN(int N, Peg target, Peg aux){
    if (N > 0){
        moveN(N-1, aux, target);
        moveTopTo(target);
        aux.moveN(N-1, target, this);
    } ●
}
```

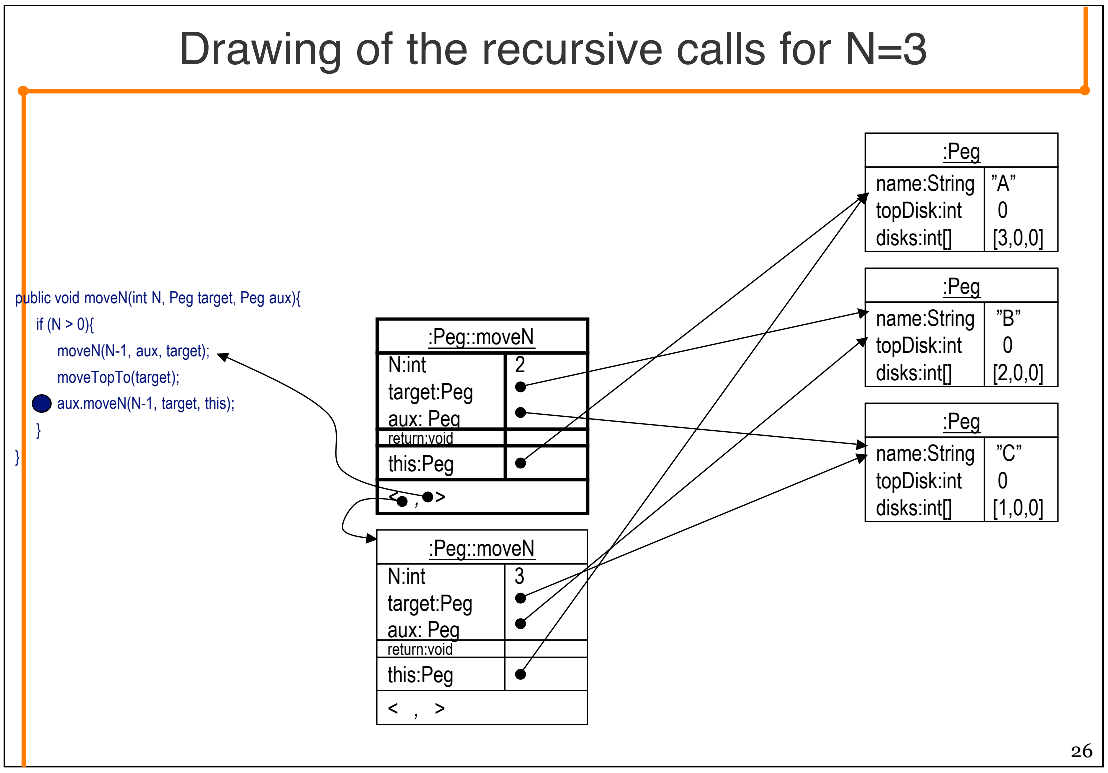
25

The first call to moveN with 0 as argument does nothing.

Neither does the last call.

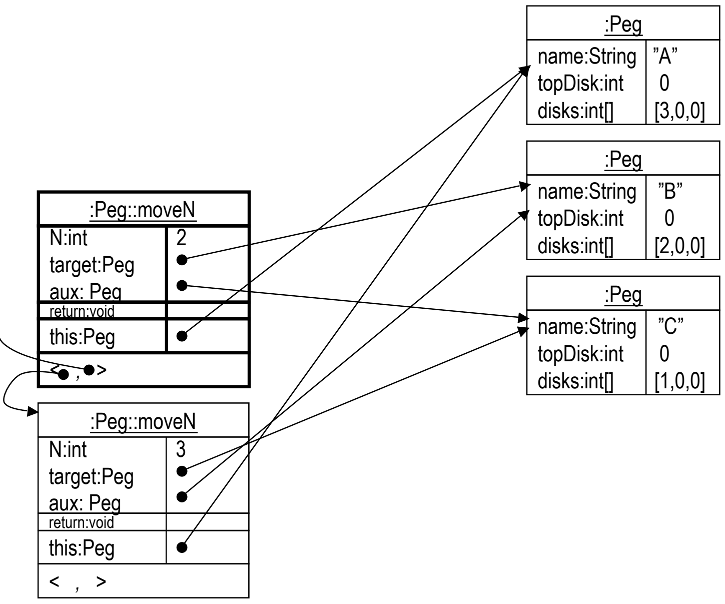
The method call moveTopTo(...) has moved the smallest disk (disk 1) to the C-Peg, leaving 2 and 3 on the A peg.

Next we are about to return from the move call. Again I indicate where in the code we are using the dot.



Drawing of the recursive calls for N=3

```
public void moveN(int N, Peg target, Peg aux){
   if (N > 0){
      moveN(N-1, aux, target);
      moveTopTo(target);
   ●  aux.moveN(N-1, target, this);
   }
}
```



26

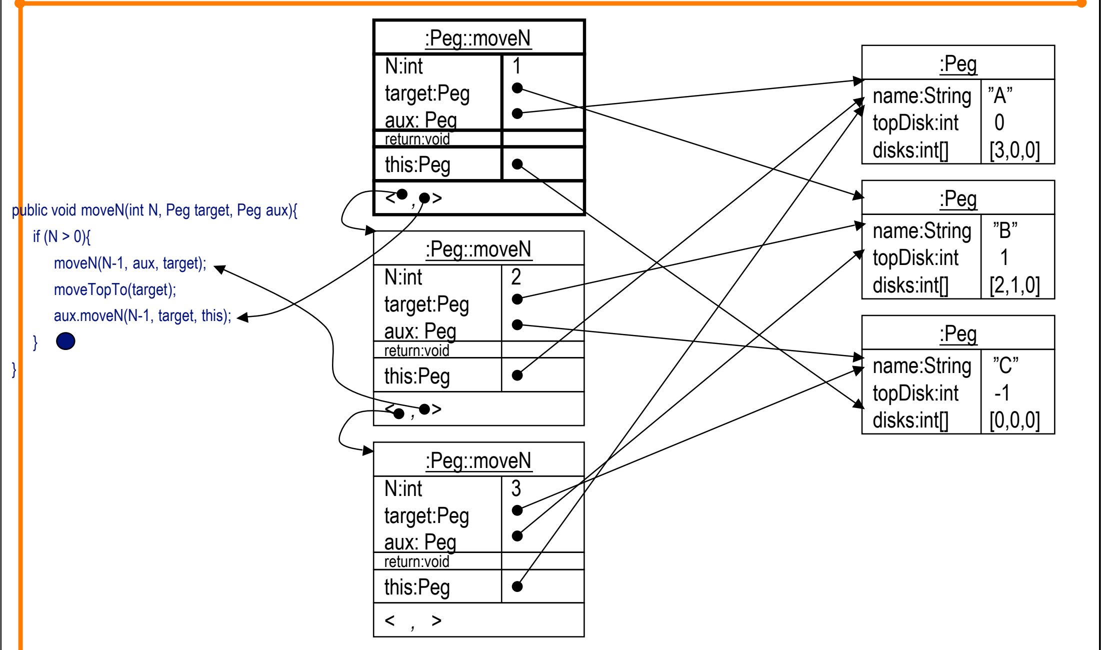
Here we have returned to the N=2 call.

We return to just after the first call of moveN.

Here I have drawn the situation as it is just before the next recursive call. The moveTopTo call moved the 2-disk to the target, which in this method call is the B-peg.

We are now ready to call the method recursively again.

Drawing of the recursive calls for N=3



```
public void moveN(int N, Peg target, Peg aux){
    if (N > 0){
        moveN(N-1, aux, target);
        moveTopTo(target);
        aux.moveN(N-1, target, this);
    }
}
```

27

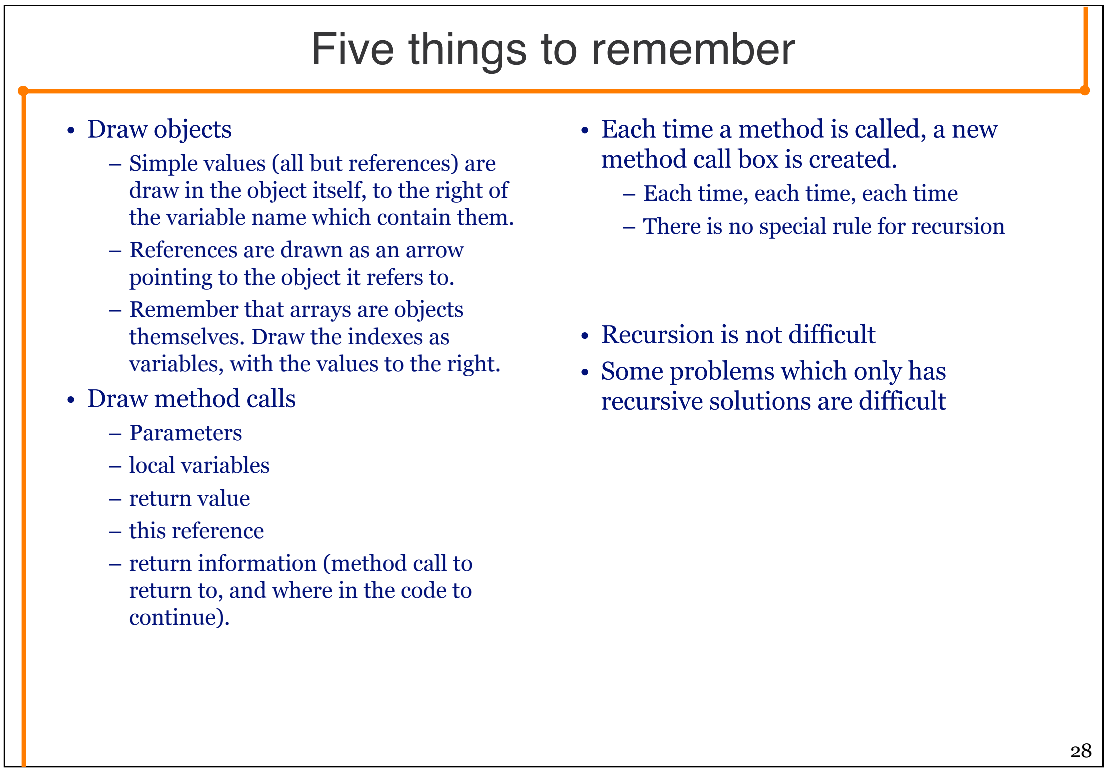
This time we called moveN on on other peg, which is reflected in the fact that the this reference is not the same.

Also observe that the return code pointer this time is different from before, as we when we return from this call return to after the last of the two recursive calls.

The drawing depicts the state of the system after we have executed the method call, and is about to return. As with all N=1 calls, there are no recursive calls that does anything. And a disk has been moved from this peg (peg-C) to the target peg (peg-B).

When we return from the N=1 call, we can see that there is no more code to be executed in the N=2 call either. And we can see that we have indeed been able to move the two topmost disks from A to B, using C as helper. The next thing to happen in N=3, is that disk 3 will be moved from A to C. Then a N=2 call will be made to move the two disks on B to C, using A as auxiliary peg.

The drawing of this is left as an exercise.



Five things to remember

- Draw objects
  - Simple values (all but references) are draw in the object itself, to the right of the variable name which contain them.
  - References are drawn as an arrow pointing to the object it refers to.
  - Remember that arrays are objects themselves. Draw the indexes as variables, with the values to the right.
- Draw method calls
  - Parameters
  - local variables
  - return value
  - this reference
  - return information (method call to return to, and where in the code to continue).
- Each time a method is called, a new method call box is created.
  - Each time, each time, each time
  - There is no special rule for recursion
- Recursion is not difficult
- Some problems which only has recursive solutions are difficult

28

But the most important thing to do now is to work with the exercises.