OOP Spring 2005
Lecture 3
Encapsulation, packages & inner classes

Kasper Østerbye
IT University Copenhagen

1

Contents

- What is encapsulation
- Member access modifiers
  - client interface
  - subclass interface
- packages
  - source files
  - scope rules
  - classpath
- Inner classes and anonymous classes
  - local classes
  - anonymous classes
  - static inner classes
  - the this reference for inner classes

Coupling and cohesion?

- If nothing depends on your class, you can change it without consequences for the rest of the system ☺. Such a class is said to be *uncoupled* from the rest of the system.
- If the members of your class does not depend on each other, you class is said to have no *cohesion*.
- When designing software, one strives to have low (not none) coupling and high cohesion.
- The hypothesis is that with low coupling, changes in one location will not propagate to the rest of the code.

Mechanisms that promote low coupling:

- private fields – renaming a field will not influence anything outside the class
- non-public classes, which can only be used inside a package

Mechanisms that enable high cohesion

- all members can be seen from inside a class

3

The goal of encapsulation is not to encapsulate the state of an object, that is, the important goal is not to make all fields private. The important goal is to design one's classes in such a way that changes to one part of the system does not influence all other parts of the system. The goal is thus to have a design that *encapsulates the impact of a change*. If in you system this is best done having all fields public and all methods private, please do it that way.

However, the following guidelines might be useful:

1. If a class A only uses the public interface of B, A and B *need not* be in the same package.
2. If A need access to some aspects of B, which should not be used by all classes, A and B might belong to the same package.
3. If A cannot exist without the presence of B, A might be better seen as an inner class of B. Making A inner in B tells other programmers that a change in B will most likely also mean changing A.

access modifiers

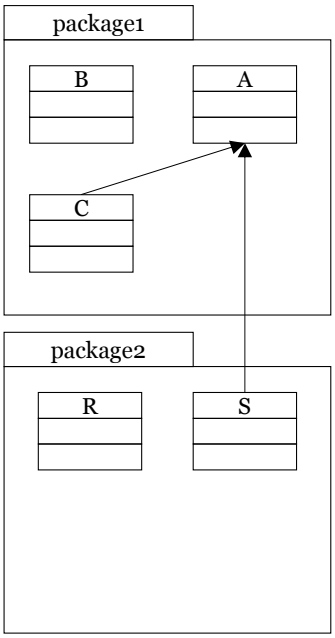
Consider the access modifier for a member x in class A. If it is:

- private – it can only be used inside A.
- default – it can be used anywhere in package1, that is, A,B and C.
- protected – it can be used anywhere in package1, and in subclasses of A in other packages, here S.
- public – it can be used everywhere in the system.

Note: an inner class I of A can access private fields of A.

Consider the access modifier for the class B. If it is:

- default – the class can only be used in package1.
- public – the class can be used from anywhere.



4

As said on the previous slide, one should not just make all fields private to make a class well encapsulated, and then write a getter and setter for all fields. On the next slide we shall examine a well encapsulated class which does not follow that common principle.

However, experience is that until one is an expert designer, it is safe to follow the guidelines:

1. make all fields private.
2. make a getter method for all fields. If necessary also make a setter method.
3. if possible, make a method private, else make it default, or make it public.

These guidelines should be balances by the following:

1. The class has a purpose, and as such must provide a certain behaviour. This behaviour must be exposed as public methods.

The access modifiers of Java are not too well designed. If the default access modifier was named ”package”, the protected modifier could be changed to mean that the member can only be seen in subclasses (but in any package). Using both package and protected, the member should only be visible in subclasses within the same package. As it is now, one cannot state that this member should only be visible in subclasses. protected means also package visibility.

Encapsulation

Consider the Person class

```
public class Person {
    int CPRnumber;
    String name;
    String address;
}
```

What access modifiers should be used, and which accessors should define?

My analysis is the following

1) The CPR number must be given when the person object is created, and cannot be changed later.
2) The Name must be given when the object is created. Normally it will not change later.
3) The address need not be present, but it can be changed along the way.

```
public class Person {
    private final int CPRnumber;
    private String name;
    private String address;

    public Person(int cpr, String name){
        CPRnumber = cpr;
        this.name = name;
    }
    public String getName(){ return name;}

    public String getAddress(){ return address;}
    public void setAddress(String address){
        this.address = address;
    }
}
```

5

Note, it is entirely possible for a person to change name in the real world. Also, when moving to a different country, that person is likely to receive an extra CPR like number, thus having two such. However, in any given system, there is a perspective defined by the purpose of the system, which should be used to delimit how the different aspects of the object should be used.

In the above, I decide that the address should have both a getter and a setter method.

Now, consider the change that we change the representation of an address from a simple string, to an object of type Address (which has such fields as streetname, streetnumber, flatnumber, postal number, city name, and country), and a USAddress as a subclass which includes also state name.

How can we encapsulate this?

First, we will assume that the Address class has a toString method, which can be used to let the getAddress() method be changed into

```
public String getAddress(){ return address.toString();}
```

However, it is not quite clear that the setAddress is easy to write, it must be able to change a purely textual representation into an address object, and it must be able to figure out if the address is a USAddress or not, and make the instance accordingly.

It is not certain that such a method can be made. Therefore this change will breake the encapsulation of the Person class.

Even if we can write the setAddress method, we might get the unexpected result that the string we give as parameter to the setAddress method is not the same as the one returned by the getAddress method. This is because the toString method in class Address might format the string differently than the one we gave as input to setAddress.

Access modifiers and inheritance

If a method is redefined in a subclass, it must be at least as visible as in the superclass.

This rule is checked when the program is compiled.

Assume the program to the right.

Consider the assignment

The variable a is declared to be of type A, and in A, the getANumber method is public. Therefore the call is legal.

But unaware to the poor compiler, a refers to an instance of B, in which the getANumber is private. UUPS.

By the rule on top, the error is in the definition in class B.

```
public class A {
  public int getANumber(){...}
  ...
}
public class B extends A {
  private int getANumber(){...}
}

public class TestAB {
  public static void main(String[] args){
    A a = new B();
    int n = a.getANumber();
  }
}
```

6

The above situation can be used to illustrate almost all known issues of academic interest in relation to single inheritance, and we will return to it many times in this course. The setup is that we have a class A and a subclass B, and a variable a of type A, which refers to an instance of type B. The general problem is that the compiler cannot know the a refers to a B-object, and must assume a refers to an A-object. This has importance in each call to a method like a.getANumber(). We shall later look at what this means for return types of the message, and for parameters and exceptions.

In the above situation, it illustrates that we cannot allow a stronger access modifier, because then we would get a runtime error. The general goal in the design of languages like Java (all object oriented languages in which one declares the type of a variable), is that method calls should succeed. One strive to make sure that errors that could be caught at compile time are indeed caught at compile time

In Java (and all other languages) we cannot normally know what kind of object a reference refers to, except that it is at least an A.

Packages

All classes belong to a package. The *default* package is used unless an other package is specified.

The name of a package is a sequence of names, separated by ”.”. For example, ”java.lang”, or ”dk.itu.oop.lecture3”.

The *fully qualified name* of a class is the name of the package followed by the a ”.”followed by the name of the class. The fully qualified name of class String is ”java.lang.String”.

A package does not declare which classes belong in it. Instead a class define which package it belong to.

This is done by the package declaration in a sourcefile. E.g.

```
package dk.itu.oop.lecture3;
```

The class Ball from lecture 1 can be used in a simple animation of a moving ball.

```
1. package dk.itu.oop.ballgame;
2. import dk.itu.oop.lecture1.Ball;
3. import java.awt.*;
4. public class MovingBall extends Ball {
5.     private final Component myComponent;
6.     private Color col;
7.     …
8. }
```

7

To use a class from an other package, one must either use the fully qualified name, or import it using an import statement.

In line 2, the class with the fully qualified name dk.itu.oop.lecture1.Ball is given the nickname Ball in this file.

In line 3, the import statement in effect says – whenever you encounter a class name in this file, and you do not know it, look in package java.awt to see if it is there. This means that we do not need to use the fully qualified name java.awt.Component in line 5, and java.awt.Color in line 6.

classpath

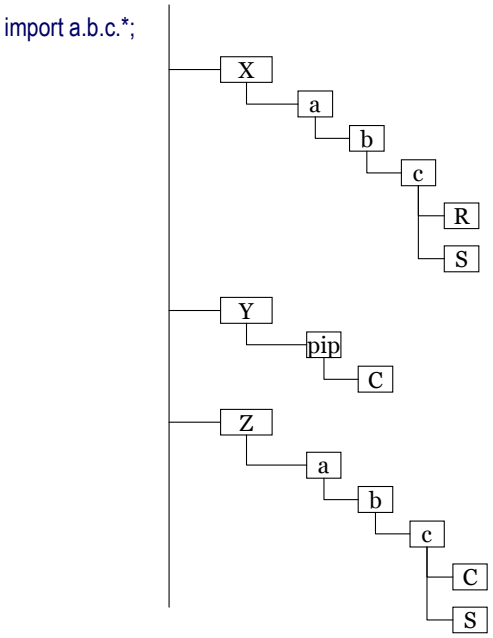
It is not specified as part of the Java language how to find all classes that belong to a package.

It is the job of a specific object in Java, known as the ”classloader” to find classes.

The standard classloader for applications use the environment variable ”classpath” to search for classes.

If the classpath variable has three directories in it, X,Y,Z, the the classloader first look for a class C in X. If it is not there, it will look in Y, and at last it will try X.

Note. It will look for a class C in package a.b.c by first looking for C in X/a/b/c, then in Y/a/b/c, and finally in Z/a/b/c.



8

The figure to the right attempts to illustate a file system, with three subdirectories, X,Y, Z included in the classpath. The directory X has a sub directory A, which again contains the subdirectoty b, which contains the subdirectory c. In c, the files R.class and S.class are located.

Similarly C.class is located in the subdirectoty of the pip directory in Y, and C.class and S.class are located int the X/a/b/c directory.

The standard class loader will in this situation see four classes in the package a.b.c, namely R, S, C, and S. It will see one class C in the package pip. Note, each directory can contain contributions to a package.

Note, the directories X,Y,Z are not part of the package names, it is the names of directories in which the classloader will look for packages.

This behaviour is how the classloader of the javac compiler and the standard virtual machine java works. The class loader for applets work differently, in that it tries to locate some files over the network, from where the url in the applet tag tells it to look.

It is possible to create a classloader yourself, which for instance looks in a database, or which does other strange stuff.

In the standard setting (javac and java), the classpath need not be the same at compile time and run time. See exercise 1.

Classpaths are a common source of magnificent frustration. Most modern software development environments (such as eclipse, JBuilder, ...) manages classpaths for you. Big improvement over doing it yourself!

Package names

Each package should have globally unique name.

There exist algorithms for this, which makes completely unreadable names like

"950365A9-5540-43a0-B28C-9899FC3BF54C"

Java uses a different approach: the web address in reverse order:

dk.itu.oop.lecture3

However, this is something which should not be taken too literal:

- java.lang –there is nowhere called lang.java
- dk.itu.oop.lecture3 does not exist on the net either.

But it is useful, readable, and likely to remain reasonable stable over a long period.

You can also name your package something like

horsens.jensen.lars.myproject

Inner classes

An inner class can be used to describe a class which is highly coupled to its outer class.

Consider the following two classes:

```
package dk.itu.oop.lecture3;
public class Point {
    private int x,y;
    public Point(int x,int y){
        this.x = x; this.y = y; }
    public int getX(){ return x;}
    public int getY(){ return y;}
    public void move(int dx, int dy){
        x+=dx;  y+=dy;  }
    public String toString(){
        return "Point(" + x + "," + y +")"; }
}
```

```
package dk.itu.oop.lecture3;
public class Line {
    private EndPoint p1, p2;
    private class EndPoint extends Point {
        public void move(int dx, int dy){
            p1.singleMove(dx,dy);
            p2.singleMove(dx,dy); }
        private void singleMove(int dx,int dy){
            super.move(dx,dy);}
        private EndPoint(Point p){
            super(p.getX(),p.getY());}
    }
    public Line(Point start, Point end){
        p1 = new EndPoint(start);
        p2 = new EndPoint(end);}
    public Point getStart(){ return p1; }
    public Point getEnd(){ return p2;}
    public String toString(){
        return "Line("+p1.toString()+","+p2.toString()+")";
    }
}
```

10

The class Point simply represents a point with two coordinates x and y. These are private, there is no setter, but their value can be read using getters. Their value can be changed using the move method. The Point class is not really important for this example, it just need to exist.

The interesting part is the inner class EndPoint of class Line. A line has two endpoints, which both works as *handles* on the line, if one moves either of the endpoints, the other endpoint is moved as well. One can argue whether this is desirable, but that is what I wanted.

The inner class EndPoint is private. That is, it cannot be used outside the class Line.

Note that EndPoint is a subclass of Point. It is quite common that inner classes are subclasses of something else. In most graphical user interfaces (GUI), based on awt or swing, the event-listener classes are inner classes that specializes say MouseAdapter.

The constructor for Line takes as arguments two Points, not EndPoints. This is because EndPoint is private. However, an EndPoint can be constructed from a Point.

Notice that Line can access the private aspects of EndPoint. That is, EndPoint can access the private members of Line (for example p1 and p2), and Line can access the private members of EndPoint (for example, its private constructor).

The move method of an EndPoint must be public, because it overrides a public method from the super class.

Testing the Point and Line class

This program follows the usual setup in which the variables p1 and p2 are of type Point (a super class), but p2 is assigned a reference to an instance of a subclass (an EndPoint).

```
package dk.itu.oop.lecture3;
public class PointLineTest {
    public static void main(String[] args){
        Point p1,p2;
        p1 = new Point(1,1);

        Line l = new Line(new Point(2,2),new Point(3,3));
        p2 = l.getEnd();

        System.out.println(p1);
        System.out.println(l);
        System.out.println();

        p1.move(5,5);
        p2.move(10,10);

        System.out.println(p1);
        System.out.println(l);
        System.out.println();
    }
}
```

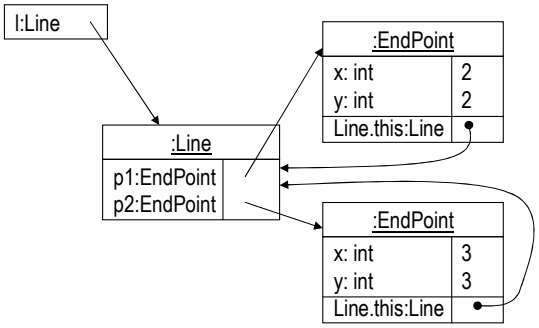
11

Inner objects and this

If we look at the classes Line and EndPoint, and an instance of a Line,

```
public class Line {
    private EndPoint p1, p2;
    private class EndPoint extends Point {
        public void move(int dx, int dy){
            p1.singleMove(dx,dy);
            p2.singleMove(dx,dy);
        }
        …
    }
    …
}
```

How can an EndPoint refer to p1 in the move method?



Just like there is an implicit this in methods, there is an implicit this in inner objects.

It can be accessed explicitly as ”Line.this”.

p1 actually means Line.this.p1.

12

Notice, just like the *this* reference in a method can not be null (remember that the compiler prevents you to use the this reference in static methods, where it is null), the *this* reference in inner objects cannot be null.

This means that an inner object cannot exist without its outer object. Sometimes an inner class is used to mirror exactly this dependency. For instance, a leg class is inner to a person, as we for most problem domains do not want loose legs.

The *this* reference is initialized when the inner object is created. If the inner object is created inside the outer object, the this reference become the outer object. This is how it is done in the example.

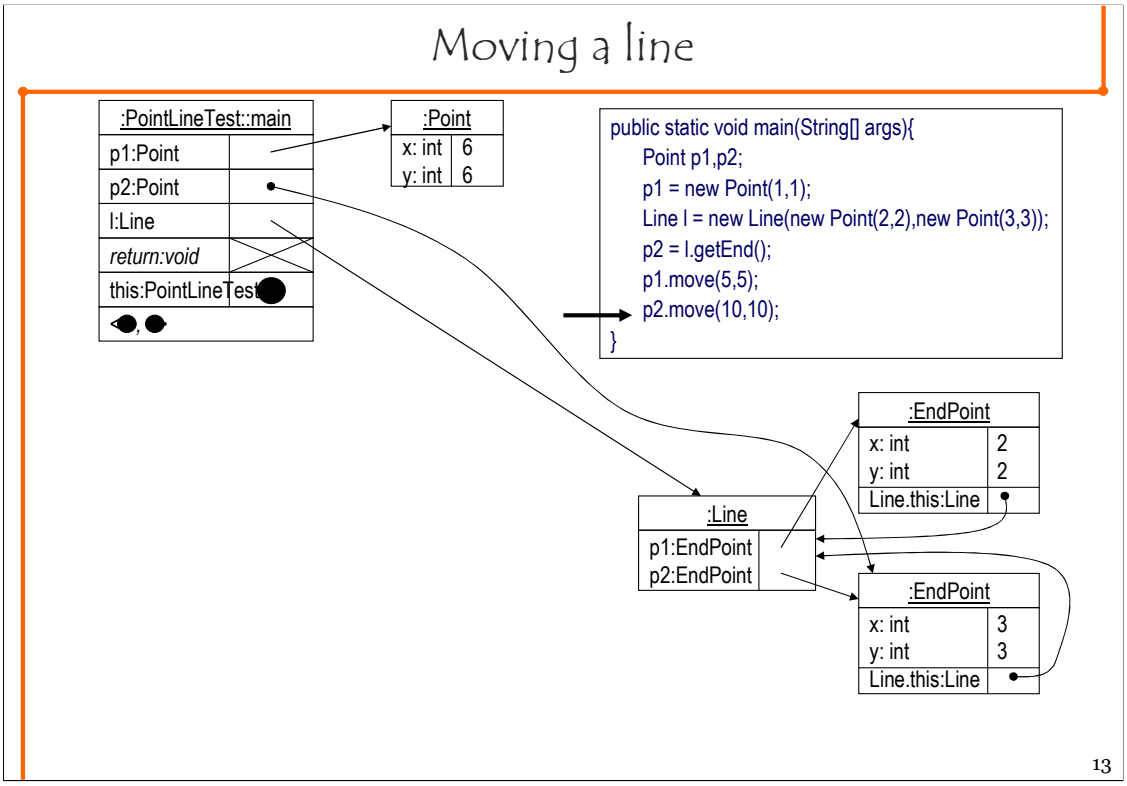
In the above Line, Point example, the inner class EndPoint is private. Sometimes it is public. If we assume it was public, then the EndPoint type can be expressed as Line.EndPoint from outside the Line class. A new inner object can then be created using the following syntax:

```
Line l = new Line(…);
Line.EndPoint lep = l.new EndPoint(…);
```

That is, we prefix the new with an object. That object become the this in the new EndPoint. Note, this means there are more then two endpoints that claim to endpoints of the line. I declared EndPoint private to avoid that.

The notion of inner classes was rediscovered by the the Java language in the mid 90ies. Inner classes was first introduced in Simula’68, and is closely related to a concept of inner methods (procedures) in a language named Algol from the early 60ies. The people who designed Java was visited by one of the designers of Beta, a successor to Simula, in which there was inner and anonymous classes.

These were the missing link for the Java designers to provide an object oriented mechanism for specifying actions in user interfaces, which today is what inner classes is used for in 95% of the cases.

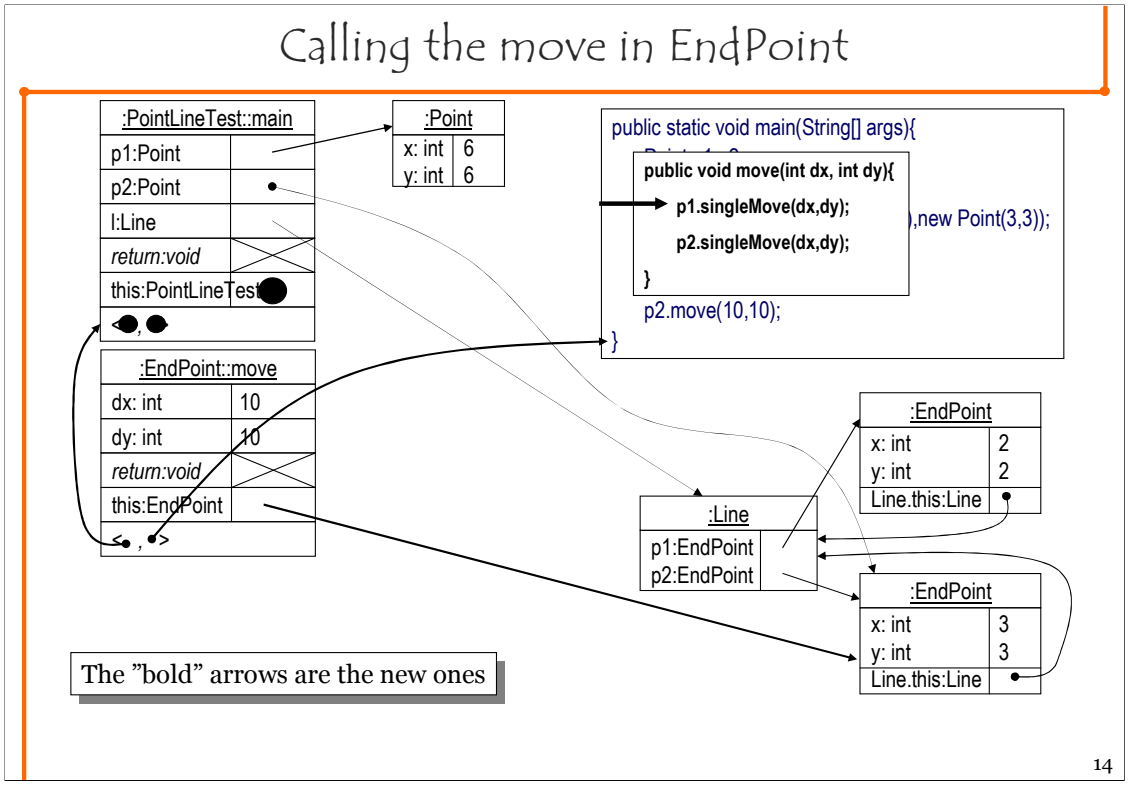


To get the operation and establishment of the this references in its place, let us consider the call to the move method in the last line above, where we move the object referenced by p2. First we can see that the variable p2 is of type Point, but refers to an object of type EndPoint.

In Java, the method to be executed is determined by the type of the object, not by the type of the reference. Thus, the call p2.move(10,10) will call the move method defined in class Endpoint.

The big fat arrow points to the next statement to be executed. The state of the objects reflect the situation just before we start that statement.

Two Point objects we created as arguments for the Line constructor. These objects are not drawn in the figure. They are not referenced anymore.



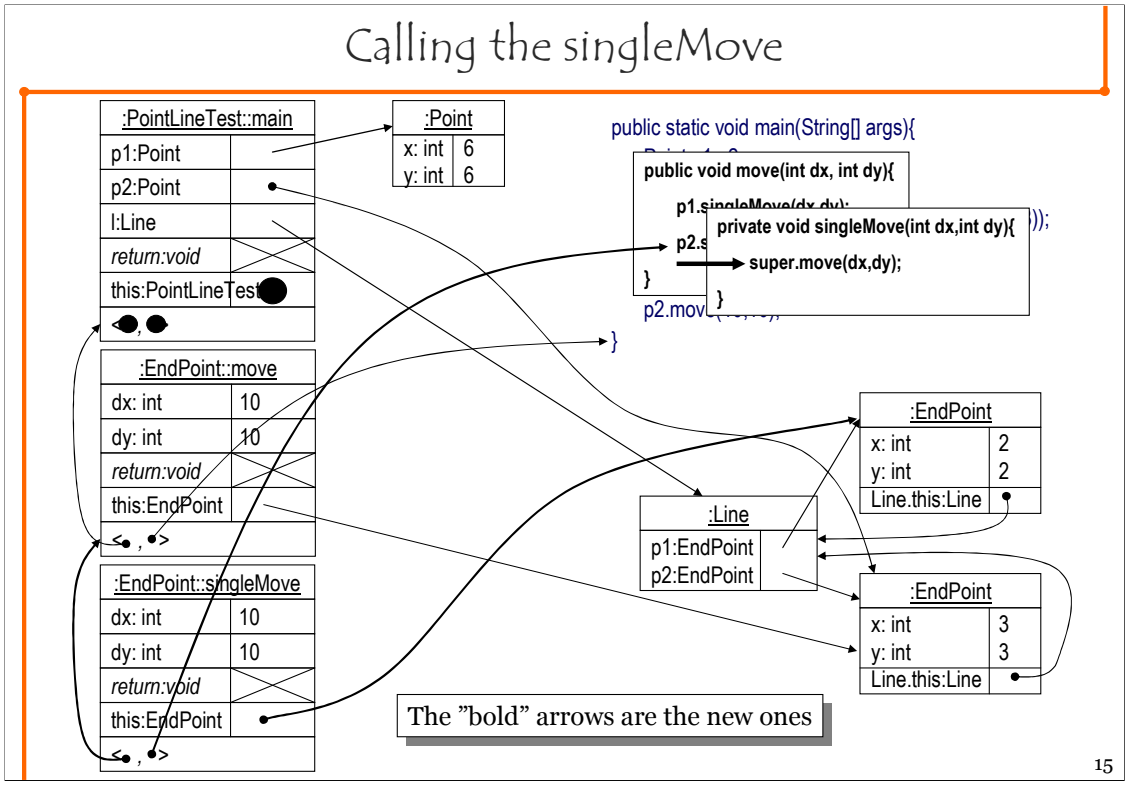
I am sorry, but this time the call stack grows downwards.

A new method call is created for storing local variables, parameters etc. There are nothing in particular of interest in this method call. The this reference refers to the endpoint which is also referred to by p2 in the main method. When we return from this method call, we return to the main method, and the next thing to be done in the main method is to return from the method call.

However, the first thing to do in the move method is to handle the call p1.singleMove(...). p1 is not defined as a local variable, in move, it is not a field in EndPoint, but it is a field in the Line class. This means that p1 is a shorthand for the reference path”this.Line.this.p1”.

Note that I have drawn a line separating x and y from Line.this in both end point objects. This is an attempt to specify that an endpoint consist of fields which come from Point, and some fields which come form the EndPoint class. In this example, there are no user defined fields in the EndPoint class. Therefore only the Line.this variable is included as a result of the subclass EndPoint.

The next thing to happen is that we call the singleMove method on the object we obtain by: a) looking in the this reference of move, b) looking in the Line.this reference of that object, c) looking in the p1 variable of the line object. This brings us to the next slide.



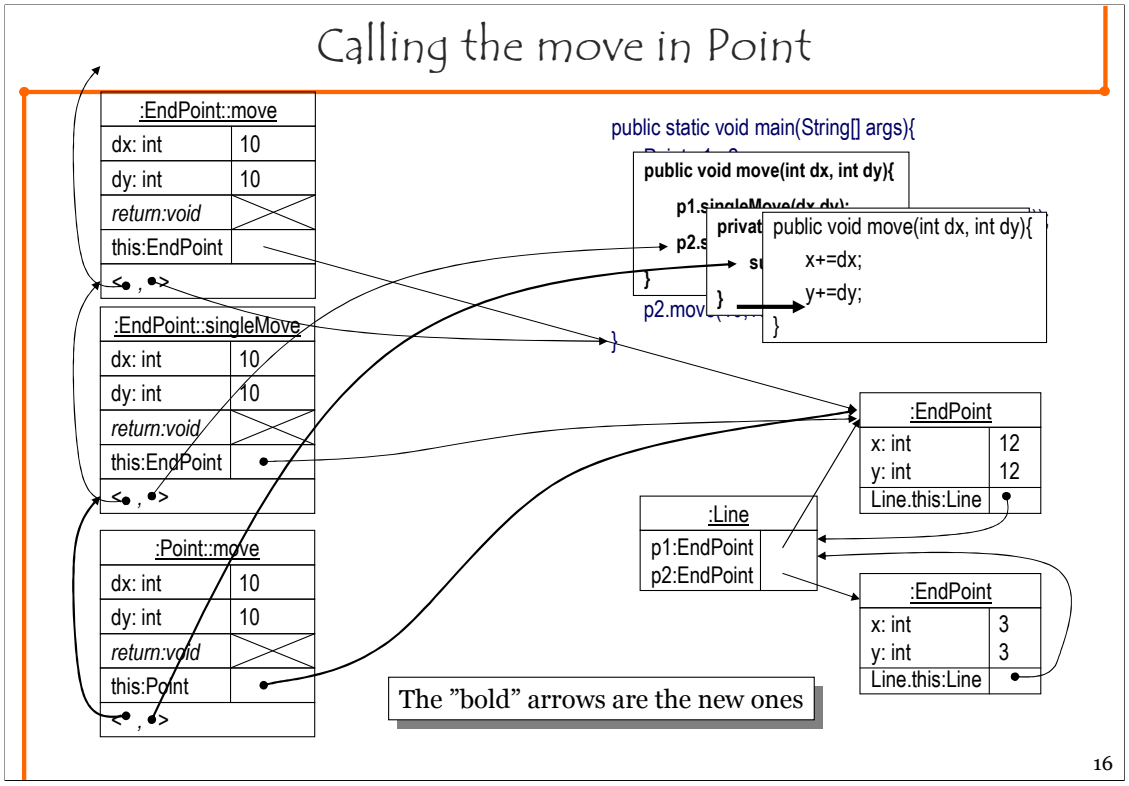
We went from the this in move(), to an Endpoint, to a Line, to an Endpoint, and we ended up calling the singleMove method on the other end point object, which is seen by the fact that the this reference in the singleMove method call refers to the other EndPoint object than does this in EndPoint::move.

The only thing singleMove does is to call super.move(). Super is explained in 9.5 of Java Precisely as a way in which to call a overridden method. Thus, super.move will call the move method as it is defined in class Point.

It would have been tempting to try to avoid the singleMove method, and try something like ”p1.super.move(...)” in the move method of EndPoint. But super cannot be used that way, it can only be used inside a subclass to refer to a overridden method.

An other attempt would have been ”((Point)p1).move(...)”. That is casting p1 to be a reference of type Point rather as EndPoint. But method calls allways use the method defined on the type of the object rather than the type of the reference, so that does not help either.

So, I introduced a method singleMove, which only moves a single point, and this method is then called from move.



Here we have entered the move method in Point. The very important thing to notice is that the this reference refers to the same object as before. The this reference always refers to the same object in connection with a super call. However, notice also that the type of the this reference has changed from EndPoint to Point.

Because this is of type Point, we can access the x and y fields (they were declared private). Had we tried to access them from a reference of type EndPoint, we would have been told that they were private, and we could not access them.

Flight example

On march 18th, SAS has a flight (SK0909) from Copenhagen to New York, Newark, scheduled to leave 12:05, and arrive 14:50. The list price for the cheapest ticket is dkr 3290,- for a round-trip ticket. The airplane to be used is an Airbus 333.

On April 18th, SK0910 is a return flight, which leaves Newark at 17:50, and arrives in Copenhagen the next morning at 7:30.

Problems:

- The same flight also leaves March 19th.
- We need to register who will man the plane.
- We need to register which seats will be free.
- We need to register the actual departure time.

```
class Flight {
    public final String flightNo;
    public final String departing, arriving;
    public final Time departureTime, arrivalTime;
    public double monkeyClassPrice;
    public final AirPlane airPlane;
    public Flight(… … …){…}
    public Time flightTime(){
        return arrivalTime.span(departureTime);
    }
}
…
Flight sk0909 = new Flight("SK0909", "CPH",
"EWR", new Time("March 18, 2004, 12:05 CET"),
new Time("March 18, 2004, 14:50 EST"), 1645,
AirPlane.get("Airbus 333"));
```

17

Flight example

The problem with the flight is common, known under the name of item-descriptor.

The descriptor here being the general description of SK0909, and the item being SK0909 on march 18th.

The solution to the right captures all Scheduled Flights. FlightSchedule captures information common to all flights, and Flight the actual flight on March 18th.

```
class FlightSchedule {
   public final String flightNo;
   public final String departing, arriving;
   public final Time departureTime, arrivalTime;
   public double monkeyClassPrice;
   public final AirPlane airPlane;
   public final Flight[] flights = new Flight[365];
   …
   class Flight {
      Date departureDate;
      Seat[] seats = new  Seat[airPlane.noSeats()];
      Time actualDeparture, actualArrival;
      …
      Time delayAtArrival(){
         return actualArrival.span(arriving);
      }
   }
}
```

18

The problem occurs in many situation, a few more examples are:

- University courses, where OOP occurs each semester, it has some common characteristics, and some things that vary from one semester to the next.
- A contract about weekly delivery of Butter to a Supermarket, and the actual weekly deliveries.
- The relationship between a book with author and title, and concrete copies with margin notes and coffee stains.

In all the examples, it is an important property that the inner class represents a weak concept, eg. a concept which has no existence without a relation to the outer class.

Note the initializer on the seats variable in Flight. The size of the array is determined as the number of Seats in the airplane. This is possible, because the airplane is a field in the outer class FlightSchedule. The outer class is known to have been initialized before any instance of a Flight is created.

Also note, the initialization of the flights array in FlightSchedule. It does not actually allocate any Flights, it just make an array to contain one flight per day in a year.

The item-descriptor problem is not always solved by the use of inner classes. There are cases where the item (the inner class) has an existence independent of the descriptor, or where more than one descriptor exist for the item. In those cases an inner class is inappropriate.

This is an example of how to initialize a flight schedule, and a flight.

```
public static void main(String[] args){
   FlightSchedule sk0909;
   FlightSchedule sk0910;

   sk0909 = new FlightSchedule
      ("SK0909", "CPH", "EWR", "12:05", "14:50",
      AirPlane.AIRBUS333);
   sk0909.flights[32] =
      sk0909.new Flight("February 1st, 2004");
   FlightSchedule.Flight sk0909Feb01 =
      sk0909.flights[32];
   sk0909Feb01.actualDeparture = "12:20";
   sk0909Feb01.actualArrival = "15:45";
}
```

19

In the final version in the code distibuted on the web page, I have changed a few types to String. This is just because Time, Date and Seat was missing, and I did not bother writing them.

Anonymous inner classes

In Java, we can make inner objects which are instances of anonymous classes.

Anonymous inner classes can be created inside methods and in connection with initializers.

Anonymous inner classes are primarily used in connection with event handling in AWT and Swing.

```
Point flipPoint = new Point(3,4){
   public void move(int dx, int dy){
            super.move(dy,dx);
   }
}

myButton.addActionListener(new ActionListener(){
   public actionPerformed(ActionEvent e){
            … do stuff …;
   }
}
```

20

Anonymous class here simply refers to the fact that the class does not have a name.

Anonymous inner classes and local variables

If we try to compile the code to the right we get the following error:

> local variable p is accessed from within inner class; needs to be declared final

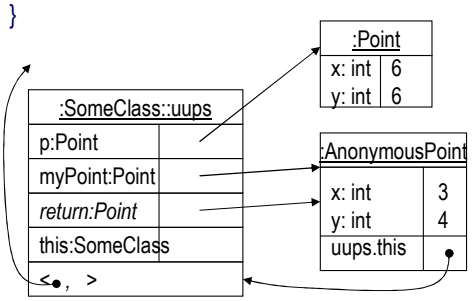
Why?

Assume, we did not get an error.

The uups.this reference is needed to get to p in the special getX method.

But when we return from the uups method, the uups method call is reclaimed, and the uups.this referece will not be valid.

```
public Point uups(Point p){
    Point myPoint = new Point(3,4){
        public int getX(){ return p.getX(); }
    };
    return myPoint;
}
```



Warning: This figure is wrong, it shows why we cannot have the code above.

21

The next couple of slides are somewhat complicated. The main thing to get hold of is *"from an inner class, one can only access local variables if they are declared as final"*

The variable p is declared outside the anonymous class. In the previous cases where a variable has been declared outside the class where it is used, the solution has been to use vaiable in the object that refered to the outer object.

Either as in the ordinary case, where the this reference gives access to the fields of an object from the methods calss of the objects (e.g. the this references in the method call instances on slide 15). Or, as we saw earlier in this lecture, where an inner object is equipped with a reference to the outer class (e.g. the . Line.this reference on slide 11)

We could be tempted to do something similar here, and make a field in the anonymous object that had a special this reference that could refere to the method call instance. Doing this would allow p.getX(), to be a shorthand for "uups.this.p.getX()".

But this has a very unattractive consequence.
In the case of this uups method, er return a reference to the anonymous point. Therefore we can use the anonymous point after we have returned from the uups method.

As we saw in lecture 2, method call instances are not reclaimed by the garbage collector, but are managed on a stack. When we return from a method call, the method call instance is recycled.

But if we used the above solution, we could not recycle the method call instance, because it is still used by the inner object.

Therefore the compiler complains, we are not allowed to refer to local variables from within a anonymous class.

The error message however, gives a hint to a solution.

Why declaring it final helps

The compiler will accept the program to the right – the only change is that the parameter p in uups is declared final.

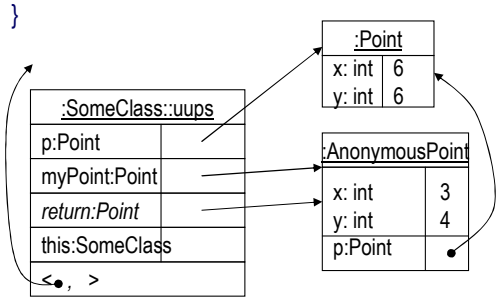
Final means that the variable can never change. This means the the variable p will for always point to p.

If a local variable is referenced from inside an inner class, the local variable must be final.

Each local variable used inside an inner class is copied to a field in the inner object.

This makes it behave as if the inner object is inner to the method call.

```
public Point uups(final Point p){
    Point myPoint = new Point(3,4){
        public int getX(){ return p.getX(); }
    };
    return myPoint;
}
```



from an inner class, one can only access local variables if they are declared as final

22

The clue in the solution is the use of final. Technically final mean that the variable will never change its value.

The surprising thing is that this helps us.

If you know that your mother knows who your grandmother is, you might as well remember it yourself, because you mother will never ever get an other mother. This is unlike the situation of a non-final variable. Your mother might have a boss at work, and you might remember it to be Johnson. But to find out who it is, you really have to go through your mother, as she might have gotten a new Job, or a new Boss.

Here we use the final as in the grandmother case. The inner object simply make a copy of the reference in the object itself.

In case it was not an object, but an integer, we can use the mother example again. If your mother was born in 1951, then that is final, her year of birth will never change. Hence, you can remember it, with out worry that one day your mother will call and say that she was born at some other time. Thus, we can use the same trick on references as well as simple types (integers, booleans etc.).

Thus, by only allowing inner classes to refer to final local variables, there exist an implementation which looks like we are really accessing the local varables, but we are not.

Remember, final means that the variable cannot change. We can still change the state of the object the variable refers to.

One more thing about anonymous inner classes

One can extend the anonymous class with methods not present in the superclass.

But one cannot call these methods.

```
Point myPoint = new Point(3,4){
    public void moveToZero(){
        super.move( -getX(), -getY() );
    }
}

myPoint.moveToZero(); // illegal call
```

23

This is quite annoying at times, especially if you have used programming languages which allow you to do so.

The reason is that myPoint is declared to be of type Point, and class Point does not have a moveToZero method.

One option could have been if myPoint was declared final, then the compiler could know that it would always refer to the inner object. But the compiler will not utilize this.

Added topic – static and final

There is a secret about classes. A class represents two kinds of objects.

1) Objects, as we have talked about them until now.
2) A singleton class object.

The singleton class object contains all the **static** members of a class description.

The singleton class object is created by the virtual machine the first time the class is used in the program.

The **final** modifier mean that the variable cannot be changed after it has gotten its initial value. The initial value must be given as an intializer or in a constructor.

As a slightly conceived example of static, is this example:

A Sir is unique in his name, that is, no two Sir's exist with the same name.

This means that the constructor should not allow two Sirs with the same name. It should throw an exception if one attempt to make a Sir with a name which already exist.

24

An important consequence of this example has to do with equality.

Because we make sure that no two instances of a Sir is created with the same name, we can always check two Sir's using ==, and be sure that if they are ==, they have the same name, and are not two different objects with the same name.

The class Sir

```
final private String name;

public Sir(String name) {
    if (find(name) != null)
        throw new Error
            ("Do not duplicate Sir " + name);
    this.name = name;
    allSirs[numberOfSirs] = this;
    numberOfSirs ++;
}

public String toString(){return "Sir " + name;}
```

```
private static Sir[] allSirs = new Sir[250];
private static int numberOfSirs = 0;
private static Sir find(String name){
    int index = 0;
    while (index < numberOfSirs){
        if (allSirs[index].name == name )
            return allSirs[index];
        index ++;
    }
    return null;
}
public static Sir getNewOrFind(String name){
    Sir sn = find(name);
    if (sn != null)
        return sn;
    else{
        sn = new Sir(name);
        return sn;
    }
}
```

25

The only state of a Sir in this example is the name. The name cannot be changed once given (final).

The class object is used to store all instances of Sirs, so we can look up and see if a Sir of a given name has already been created. This is done using two fields, an Array which contains up to 250 Sirs, and the number of instances in that array, numberOfSirs.

The method find on the class object examines if there already exist a Sir of the given name, and if so, returns that Sir, or else return null.

The constructor checks to see if we are trying to make a Sir who already exists. If we do so, an Error is thrown.

As an alternative to using the constructor, the method getNewOrFind will either return an existing Sir of a given name, or create a new Sir of that name.

The Program below shows how the Sir class might be used:

```
class SirTest {
    public static void main(String[] args){
        Sir sn1 = Sir.getNewOrFind("John");
        Sir sn2 = Sir.getNewOrFind("Robert");
        Sir sn3 = Sir.getNewOrFind("John");
        System.out.println("s1 == sn2: " + (sn1 == sn2));
        System.out.println("s1 == sn3: " + (sn1 == sn3));
        try{ sn2 = new Sir("John");
        }catch(Error e){ System.out.println(e);}
    }
}
```

It results in the following output:

```
s1 == sn2: false
s1 == sn3: true
java.lang.Error: Do not duplicate Sir John
```