OPI
Lecture 16
Encapsulation, packages & inner classes

Kasper Østerbye

Carsten Schuermann

IT University Copenhagen

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

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.
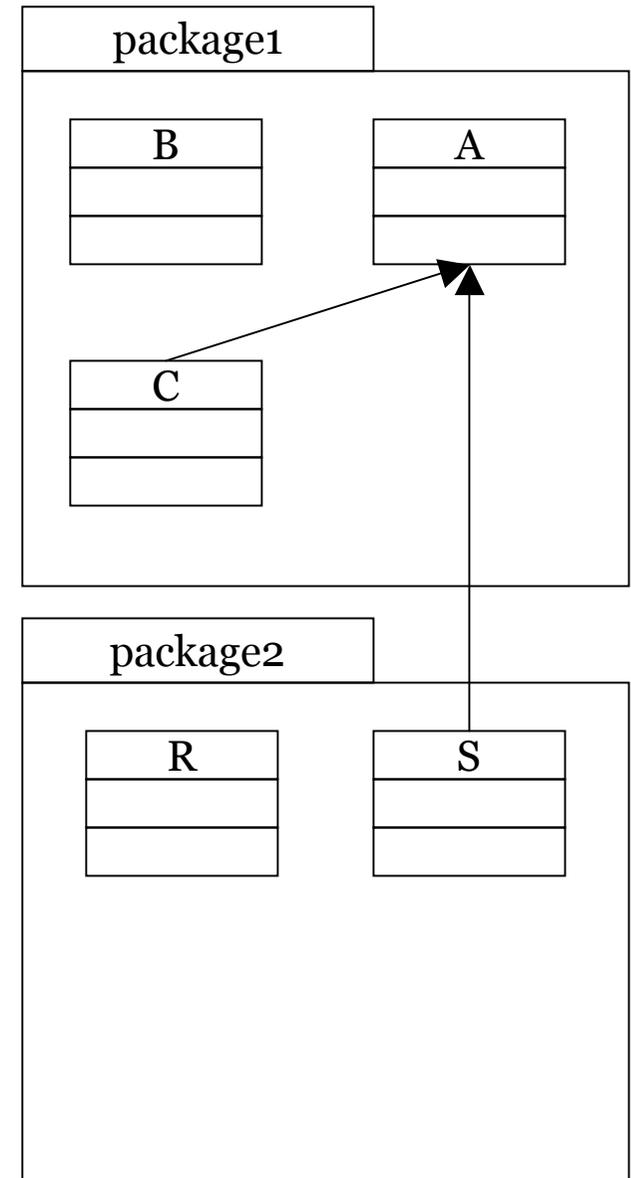
Encapsulation

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;
    }
}
```

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();
    }
}
```

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. }
```

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.

import a.b.c.*;

```
X
  a
    b
      c
        R
        S
Y
  pip
    C
Z
  a
    b
      c
        C
        S
```
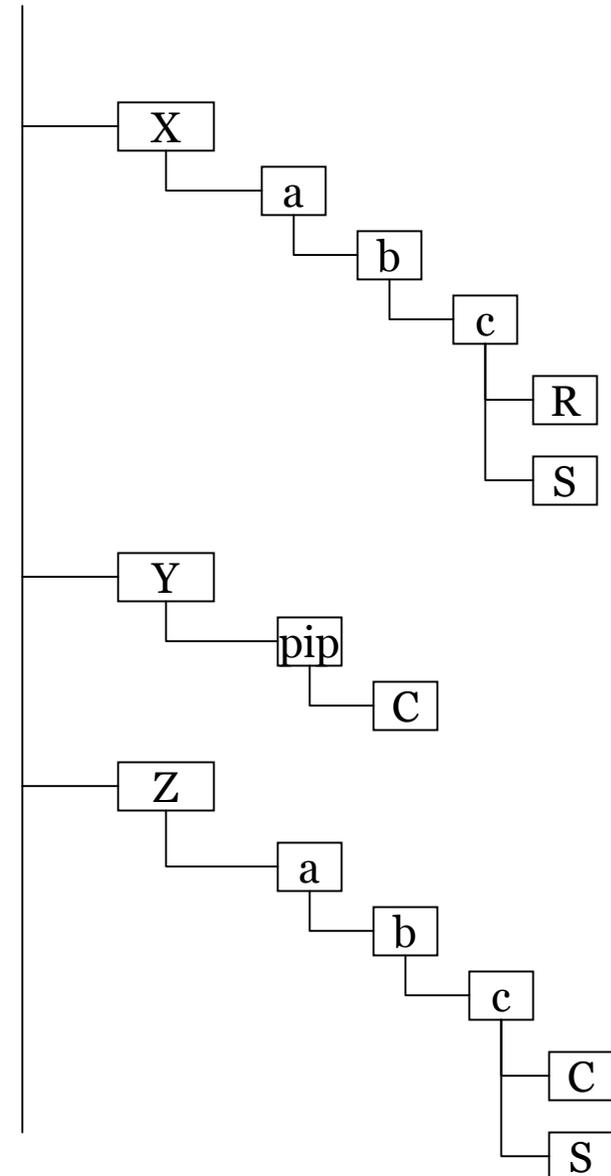
Package names

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:

```java
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 +")"; }
}
```

```java
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()+")";
    }
}
```

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();
    }
}
```

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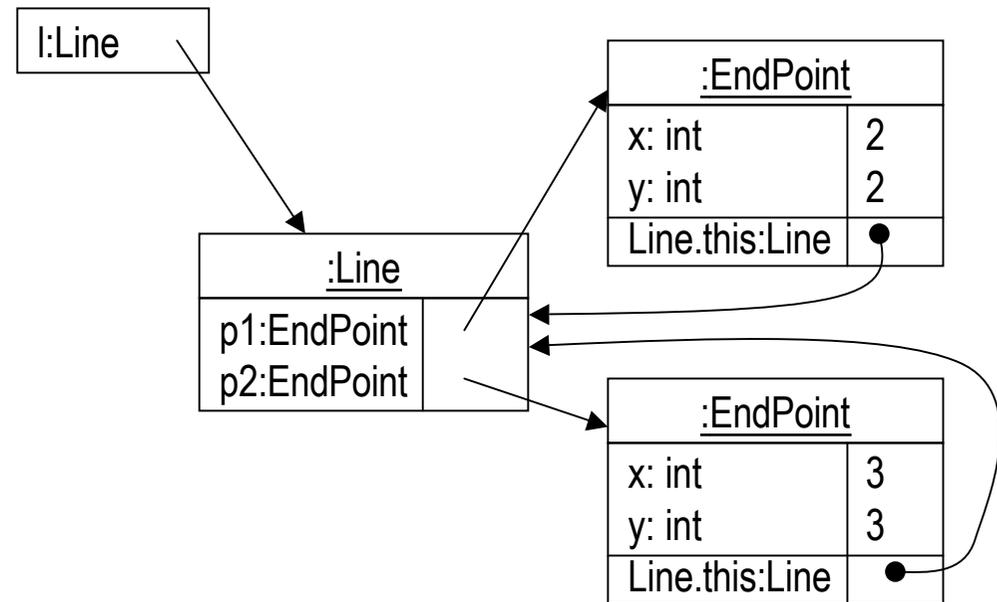
Just like there is an implicit this in methods, there is an implicit this in inner objects.

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

p1 actually means Line.this.p1.

Moving a line
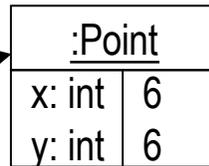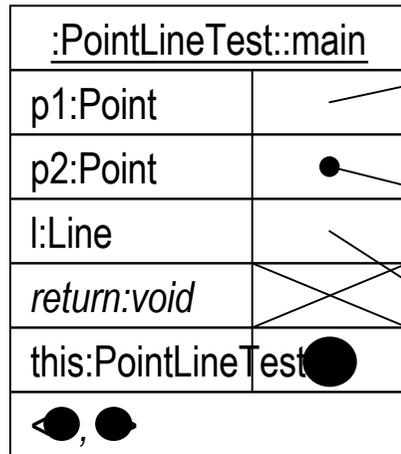
:PointLineTest::main	
p1:Point	
p2:Point	●
l:Line	
return:void	✕
this:PointLineTest	●
◄●,●	

:Point	
x: int	6
y: int	6

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

:EndPoint	
x: int	2
y: int	2
Line.this:Line	●

:Line	
p1:EndPoint	
p2:EndPoint	

:EndPoint	
x: int	3
y: int	3
Line.this:Line	●

13
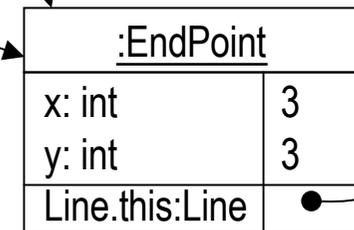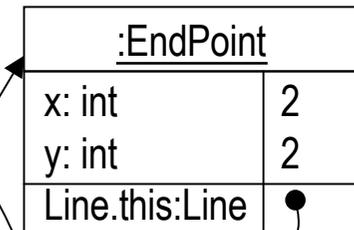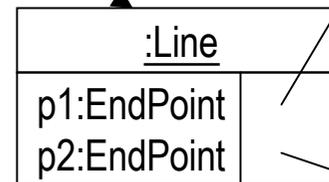
Calling the move in EndPoint

:PointLineTest::main

p1:Point	
p2:Point	●
l:Line	
return:void	✕
this:PointLineTest	●
◄●,●	

:Point

| x: int | 6 |
| y: int | 6 |

```
public static void main(String[] args){

    public void move(int dx, int dy){

        p1.singleMove(dx,dy);               ,new Point(3,3));

        p2.singleMove(dx,dy);

    }
    p2.move(10,10);

}
```

:EndPoint::move

dx: int	10
dy: int	10
return:void	✕
this:EndPoint	
◄● , ●►	

:EndPoint

x: int	2
y: int	2
Line.this:Line	●

:Line

| p1:EndPoint | |
| p2:EndPoint | |

:EndPoint

x: int	3
y: int	3
Line.this:Line	●

The "bold" arrows are the new ones

:PointLineTest::main

p1:Point	
p2:Point	
l:Line	
return:void	
this:PointLineTest	

:Point

| x: int | 6 |
| y: int | 6 |

```
public static void main(String[] args){

    public void move(int dx, int dy){

        p1.singleMove(dx,dy);
                            private void singleMove(int dx,int dy){    ));

        p2.s                    super.move(dx,dy);

    }                    }

}                    }
    p2.move(...);
}
```

:EndPoint::move

dx: int	10
dy: int	10
return:void	
this:EndPoint	
<● , ●>	

:EndPoint::singleMove

dx: int	10
dy: int	10
return:void	
this:EndPoint	●
<● , ●>	

:EndPoint

x: int	2
y: int	2
Line.this:Line	●

:Line

| p1:EndPoint | |
| p2:EndPoint | |

:EndPoint

x: int	3
y: int	3
Line.this:Line	●

The "bold" arrows are the new ones

Calling the move in Point

:EndPoint::move

dx: int	10
dy: int	10
return:void	✕
this:EndPoint	
<● , ●>	

:EndPoint::singleMove

dx: int	10
dy: int	10
return:void	✕
this:EndPoint	●
<● , ●>	

:Point::move

dx: int	10
dy: int	10
return:void	✕
this:Point	●
<● , ●>	

```
public static void main(String[] args){
```
```
public void move(int dx, int dy){

    p1.singleMove(dx,dy);

    p2.s

}
```
```
privat    public void move(int dx, int dy){

    su        x+=dx;

}           y+=dy;

p2.mov        }
}
```

:EndPoint

x: int	12
y: int	12
Line.this:Line	●

:Line

p1:EndPoint	
p2:EndPoint	

:EndPoint

x: int	3
y: int	3
Line.this:Line	●

The "bold" arrows are the new ones

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"));
```

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);
        }
    }
}
```

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

```java
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";
}
```

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 …;
    }
}
```

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;
}
```
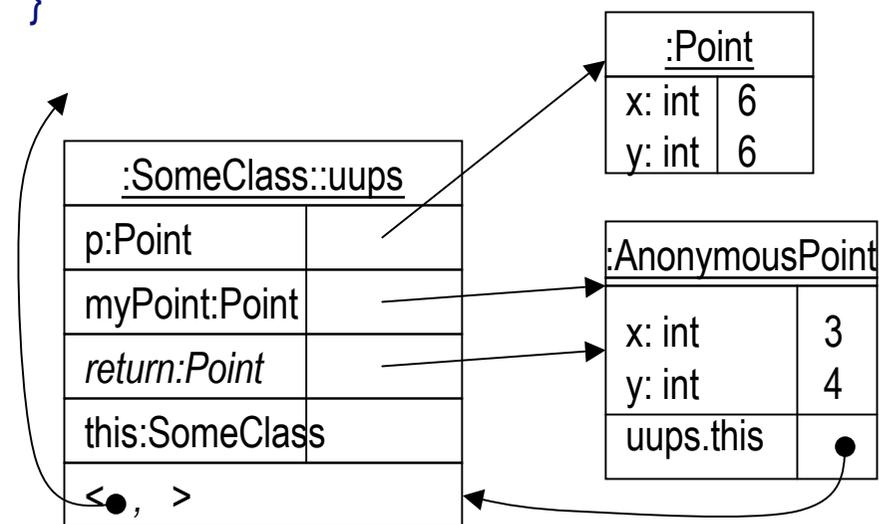

Warning: This figure is wrong, it shows why we cannot have the code above.

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;
}
```

:Point	
x: int	6
y: int	6

:SomeClass::uups	
p:Point	
myPoint:Point	
return:Point	
this:SomeClass	
<● , >	

:AnonymousPoint	
x: int	3
y: int	4
p:Point	●
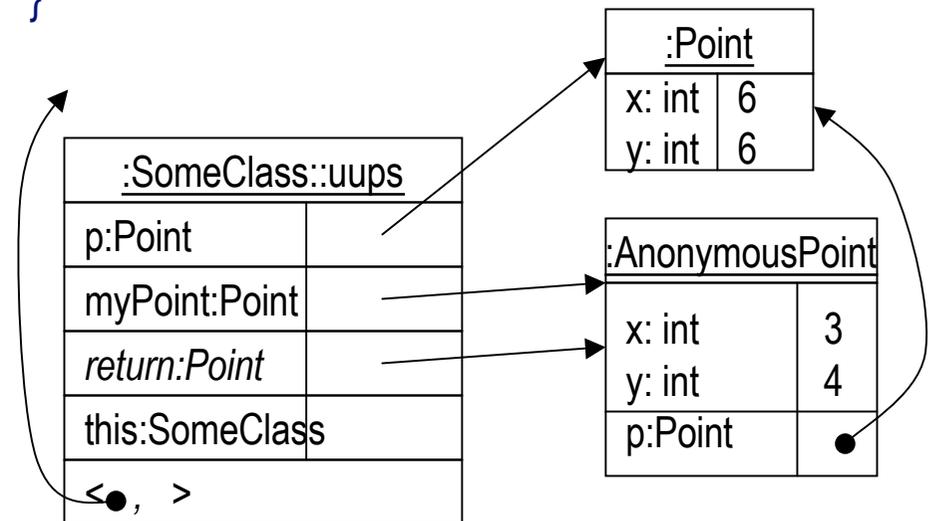
from an inner class, one can only access local variables if they are declared as final

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
```

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.

The class Sir

```java
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;}
```

```java
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;
    }
}
```