Object composition

Class `Point`

```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point(Point p) {
        /* requires p!=null; ensures this.x=p.x && this.y==p.y; */
        this(p.x, p.y);
    }
    public int getX() { return this.x; }
    public int getY() { return this.y; }
    public void move(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }
    public boolean overlaps(Point p) {
        /* requires p!=null; ensures result==(this.x==p.x && this.y==p.y); */
        return this.x == p.x && this.y == p.y;
    }
}
```

Object composition

Class `Line` badly designed

```
public class Line {
    private Point a;
    private Point b;
    /* invariant a != null && b != null && !a.overlaps(b); */
    public Line(Point a, Point b) {
        /* requires a!=null && b!=null && !a.overlaps(b); */
        if (a.overlaps(b))
            throw new IllegalArgumentException();
        this.a = a;
        this.b = b;
    }
    public void move(int dx, int dy) {
        this.a.move(dx, dy);
        this.b.move(dx, dy);
    }
    public boolean overlaps(Line l) { /* requires l!=null; */
        return this.a.overlaps(l.a) && this.b.overlaps(l.b)
                || this.a.overlaps(l.b) && this.b.overlaps(l.a);
    }
}
```

Object composition

Problem with class `Line`

`l2.overlaps(l3)` is no longer **`true`** after calling `l1.move(1, 0)`

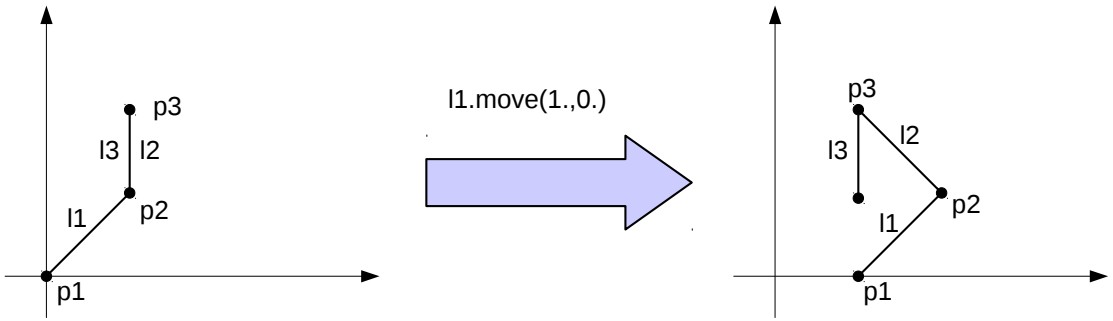
Demo

```
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
Point p3 = new Point(1, 2);
Line l1 = new Line(p1, p2);
Line l2 = new Line(p2, p3);
Line l3 = new Line(new Point(p2), new Point(p3));
assert l2.overlaps(l3);
l1.move(1, 0);
assert !l2.overlaps(l3);
```

Solution to the problem

The problem

- private point components can be modified from the client code
- moving a point or a line may have the side effect of moving other lines
- reasoning on a program with points and lines becomes quite difficult



Solution: exclusive ownership

- a line segment must exclusively own its two end points:
  - do not allow end points to be modified from the client code
  - do not allow end points to be shared with other lines
- in the constructor of `Line` point arguments must be copied

Revisited code

Class `Line` correctly designed

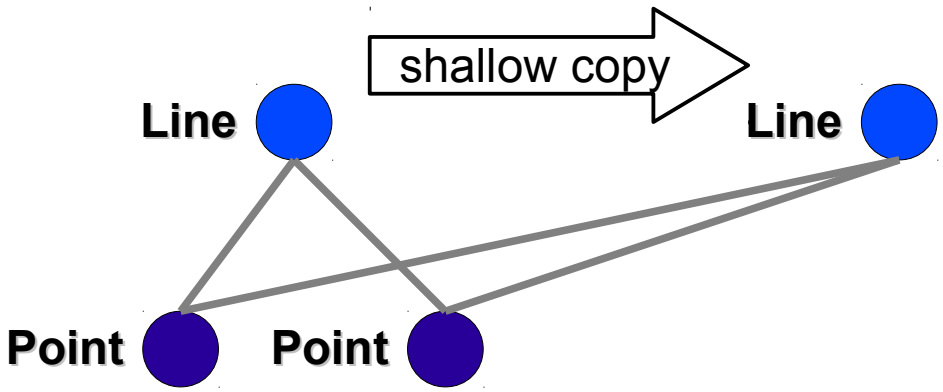
```
public class Line {
    private Point a;
    private Point b;

    // invariant a != null && b != null && !a.overlaps(b)
    public Line(Point a, Point b) {
        /* requires a!=null && b!=null && !a.overlaps(b); */
        if (a.overlaps(b))
            throw new IllegalArgumentException();
        this.a = new Point(a); // a new copy of a
        this.b = new Point(b); // a new copy of b
    }
    public void move(int dx, int dy) {
        this.a.move(dx, dy);
        this.b.move(dx, dy);
    }
    public boolean overlaps(Line l) { /* requires l!=null; */
        return this.a.overlaps(l.a) && this.b.overlaps(l.b)
                || this.a.overlaps(l.b) && this.b.overlaps(l.a);
    }
}
```

Revisited code

The test now works as expected!

```
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
Point p3 = new Point(1, 2);
Line l1 = new Line(p1, p2);
Line l2 = new Line(p2, p3);
Line l3 = new Line(new Point(p2), new Point(p3));
assert l2.overlaps(l3);
l1.move(1, 0);
assert l2.overlaps(l3); // ok, moving l1 does not affect l2
```

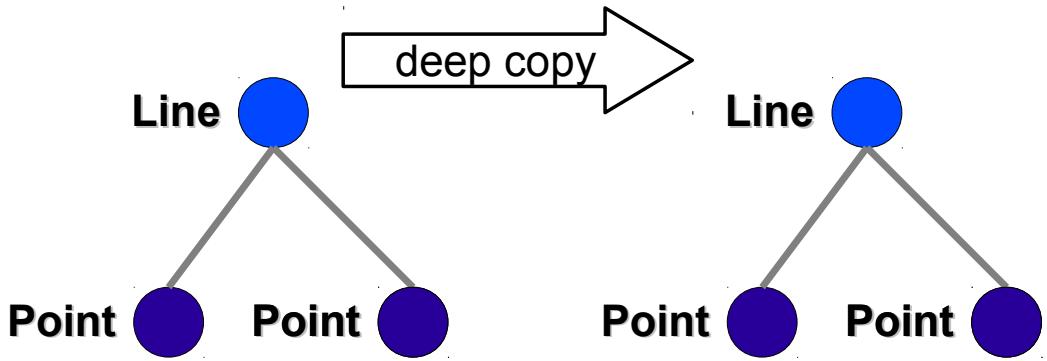
Shallow and deep copy



Shallow copy

```
public Line(Line l) {
// no exclusive ownership!
    this.a = l.a;
    this.b = l.b;
}
```

Shallow and deep copy



Deep copy

```
public Line(Line l) {
    this.a = new Point(l.a);
    this.b = new Point(l.b);
}
```

Final variables in Java

Rules

- fields, local variables and parameters can be declared final
- a final variable is read-only: it always contains the same value

Remark

If a final variable refers to an object, then

- it will always refer to the same object
- but that object could be modified, if it is modifiable

Initialization of final fields

- a final object field must be initialized as follows:
  - either with a field initializer and in no other ways
  - or with every constructor of its class and in no other ways
- a final class field must be initialized as follows:
  - either with a field initializer and in no other ways
  - or with a single static initializer of its class and in no other ways

Final variables in Java

Example

```
public class Item {
    private static long availableSN;
    private int price;
    public final long serialNumber; // 'serialNumber' is constant
    public Item(int price) {
       ...
    }
    public int getPrice() {
        return this.price;
    }
    public long getSerialNumber() {
        return this.serialNumber;
    }
}
```

Remark

Since `serialNumber` is constant, there is no arm if it is public

Final variables in Java

Example

```
public class Rectangle {
   public static final int defaultSize = 1; // 'defaultSize' is constant
   private int width = Rectangle.defaultSize;
   private int height = Rectangle.defaultSize;
   public Rectangle(int width, int height) {
       ...
   }
   public static int getDefaultSize() {
       return Rectangle.defaultSize;
   }
   public int getWidth() {
       return this.width;
   }
   public int getHeight() {
       return this.height;
   }
}
```

Remark

Since `defaultSize` is constant, there is no arm if it is public

Mutable versus immutable objects

Class `Line` with final fields

```
public class Line {
    private final Point a;
    private final Point b;
    public Line(Point a, Point b) {
        if (a.overlaps(b))
            throw new IllegalArgumentException();
        this.a = new Point(a);
        this.b = new Point(b);
    }
    public void move(int dx, int dy) {
        this.a.move(dx, dy);
        this.b.move(dx, dy);
    }
    public boolean overlaps(Line l) {
        return this.a.overlaps(l.a) && this.b.overlaps(l.b)
                || this.a.overlaps(l.b) && this.b.overlaps(l.a);
    }
}
```

Question

Are objects of class `Line` immutable?

Mutable versus immutable objects

Answer

Are objects of class `Line` immutable? No!

- the end points of a line will always be the same objects

But:

- the state of a line depends on the state of its end points
- the end points of a line are mutable $\Rightarrow$ the line is mutable as well

Mutable versus immutable objects

Sufficient conditions for an object to be immutable

- all object fields are final

  and
- each field contains
  - either a primitive value (a number or a boolean value)
  - or an immutable object

A field can be safely declared `public` if

- it is final

  and
- it contains
  - either a primitive value (a number or a boolean value)
  - or an immutable object

  and its associated information can be public

Interfaces

A motivating example

```
public class TimerClass {
    private int time = 60;
        ...
    public TimerClass(TimerClass other) {
        this.time = other.getTime();
    }
        ...
}
public class AnotherClass {
    private int minutes = 1;
    private int seconds;
        ...
    public AnotherTimerClass(AnotherTimerClass other) {
        int time = other.getTime();
        this.minutes = time / 60;
        this.seconds = time % 60;
    }
        ...
}
```

Interfaces

A motivating example

```
//this code contains two static errors
TimerClass t1 = new TimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass(t2); //error 1:  AnotherTimerClass ⋠ TimerClass
AnotherTimerClass t4 =
    new AnotherTimerClass(t1); //error 2:  TimerClass ⋠ AnotherTimerClass
```

Problem

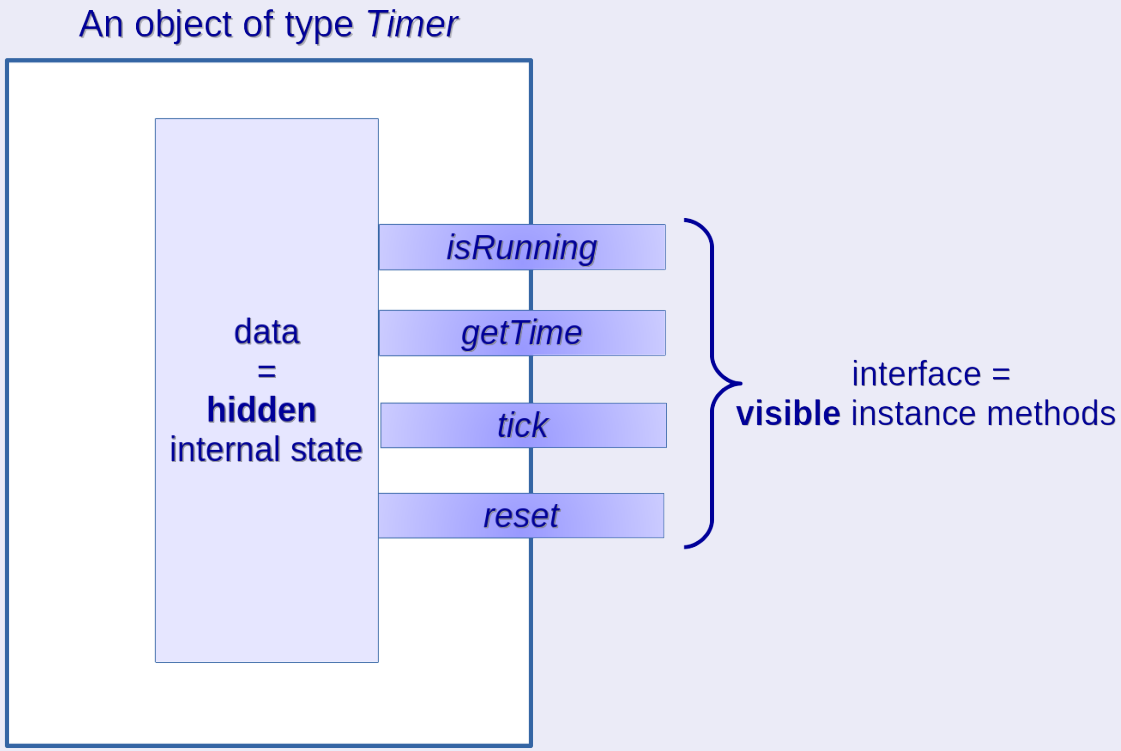
Timers of type `TimerClass` and `AnotherTimerClass` are not comparable

Solution

- declare the parameter `other` with the more general type `Timer`
- `TimerClass` $\leq$ `Timer` and `AnotherTimerClass` $\leq$ `Timer`

Interfaces

Type `Timer` is the object interface



Interfaces

Definition of `Timer` in Java

```
public interface Timer { // Timer is a type but not a class
    // all these methods are abstract and public
    boolean isRunning();
    int getTime();
    void tick();
    int reset(int minutes);
}
```

Interfaces

Solution

```
public class TimerClass implements Timer { // TimerClass ≤ Timer
    private int time = 60;
        ... // all methods of Timer must be defined in the class
    public TimerClass(Timer other) {
        this.time = other.getTime();
    }
        ...
}

public class AnotherTimerClass implements Timer { // AnotheTimerClass ≤ Timer
    private int minutes = 1;
    private int seconds;
        ... // all methods of Timer must be defined in the class
    public AnotherTimerClass(Timer other) {
        int time = other.getTime();
        this.minutes = time / 60;
        this.seconds = time % 60;
    }
        ...
}
TimerClass t1 = new TimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass(t2); // no error: AnotherTimerClass ≤ Timer
AnotherTimerClass t4 = new AnotherTimerClass(t1); // no error: TimerClass ≤ Timer
```

Interfaces

A wrong solution

```
public TimerClass(Object other) {
    this.time = other.getTime() // static error
}

public AnotherTimerClass(Object other) {
    int time = other.getTime(); // static error
    this.minutes = time / 60;
    this.seconds = time % 60;
}
```

Remark: Objects of type `Object` do not have method `getTime()`

Interfaces

Details

- interfaces are useful abstractions in statically typed languages
- a class can implement more interfaces
- interfaces are more abstract than classes
- the most useful components of an interface are instance methods that are public and abstract
- the following implicit assumptions hold on instance methods of Java interfaces:
  - **`public`** can be omitted
  - **`abstract`** can be omitted, if the method has no body

Remarks

- interfaces cannot be used for creating objects, they are just types
- interfaces cannot declare constructors
- a class must define all methods of the implemented interfaces
- if class C implements interface I, then C is subtype of I ($C \leq I$)