Object composition

Example of failure

```
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);
```

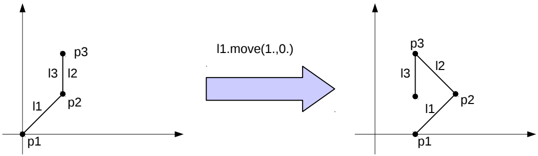
Remark

- `l2.overlaps(l3)` is no longer **`true`** after calling `l1.move(1, 0)`
- what should be the expected behavior?

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:
  - the two end points cannot be modified from the client code
  - the two end points cannot be shared with other lines
- in the constructor of `Line` points must be copied

Revisited code

Lines with exclusive ownership of points

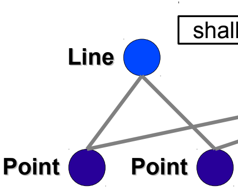
```
public class Line {
    private Point a;
    private Point b;

    // invariant a != null && b != null && !a.overlaps(b)
    public Line(Point a, Point b) {
        if (a.overlaps(b)) /* if a==null||b==null then NullPointerException is
             thrown! */
            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) {
        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
```

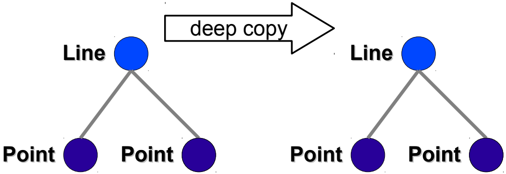
Shallow and deep copy



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

- instance/class/local variables (and parameters) can be declared **final**
- a final variable is read-only: it always contains the same value

Remark

If a variable refers to an object

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

Initialization of final instance/class variables

- a final instance variable must be initialized as follows:
  - either with a variable initializer (and in no other ways)
  - or with every constructor of its class (and in no other ways)
- a final class variable must be initialized as follows:
  - either with a variable 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; // long final variable can be public
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;
        this.serialNumber = Item.availableSN++;
    }
    public int getPrice() {
        return this.price;
    }
    // getSerialNumber() no longer needed
}
...
Item2 item1 = new Item2(61_50);
Item2 item2 = new Item2(14_00);
assert item1.getPrice() == 61_50 && item1.serialNumber == 0;
assert item2.getPrice() == 14_00 && item2.serialNumber == 1;
```

Final variables in Java

Example

```
public class Rectangle {
   public static final int defaultSize = 1; // int final variable can be public
   private int width = Rectangle.defaultSize;
   private int height = Rectangle.defaultSize;

   private static void checkSize(int size) {
      if (size <= 0)
         throw new IllegalArgumentException();
   }
   public Rectangle(int width, int height) {
       Rectangle.checkSize(width);
       Rectangle.checkSize(height);
       this.width = width;
       this.height = height;
   }
   public static Rectangle ofWidthHeight(int width,int height) {
       return new Rectangle(width, height);
   }
}
```

Mutable versus immutable objects

Example

```
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 `Line` objects immutable?

Mutable versus immutable objects

Answer

Are `Line` objects 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 ⇒ the line is mutable as well

Mutable versus immutable objects

Sufficient conditions for an object to be immutable

- all instance variables are final

  and
- each instance variable contains
  - either a primitive value (not an object)
  - or an immutable object

A class/instance variable can be safely declared `public` if

- it is final

  and
- it contains
  - either a primitive value (not an object)
  - or an immutable object

Interfaces

A motivating example

```
public class TimerClass {
    private int time = 60;
        ...
    public TimerClass(TimerClass otherTimer) {
        this.time = otherTimer.time;
    }
        ...
}
public class AnotherTimerClass {
    private int minutes = 1;
    private int seconds;
        ...
    public AnotherTimerClass(AnotherTimerClass otherTimer) {
        this.seconds = otherTimer.seconds;
        this.minutes = otherTimer.minutes;
    }
        ...
}
```

Interfaces

A motivating example

```
TimerClass t1 = new TimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass(t2); // error:  AnotherTimerClass ≰ TimerClass
AnotherTimerClass t4 =
    new AnotherTimerClass(t1); // error:  TimerClass ≰ AnotherTimerClass
```

Problem

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

Possible solution

- use getter `getTime()`
- use a supertype of `TimerClass` and `AnotherTimerClass`

Definition

C_1 is supertype of C_2 if and only C_2 is subtype of C_1

Interfaces

A wrong solution

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

Error

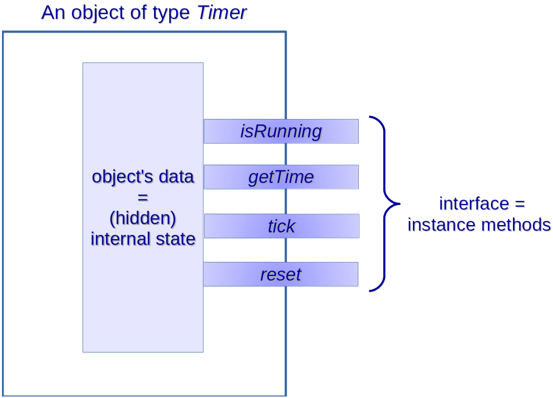
- `TimerClass` $\leq$ `Object`
- `AnotherTimerClass` $\leq$ `Object`

  but
- Objects of type `Object` do not have method `getTime()`!

Interfaces

Desiderata

- Define a supertype of `TimerClass` and `AnotherTimerClass`
- with `getTime()` and all other instance methods of the Timer interface



Interfaces

Example

```
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);
}
```

Remark

Interfaces cannot contain constructors

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 otherTimer) {
        this.time = otherTimer.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 otherTimer) {
        int time  = otherTimer.getTime();
        this.minutes = time / 60;
        this.seconds = time % 60;
    }
        ...
}
```

Interfaces

Details

- interfaces are useful abstractions in statically typed OOL (Java, C#, TypeScript, Kotlin)
- a class can implement more interfaces
- interfaces are more abstract than classes
- all methods in an interface are implicitly
  - **`public`**
  - **`abstract`**: they contain no body!
  - instance methods

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