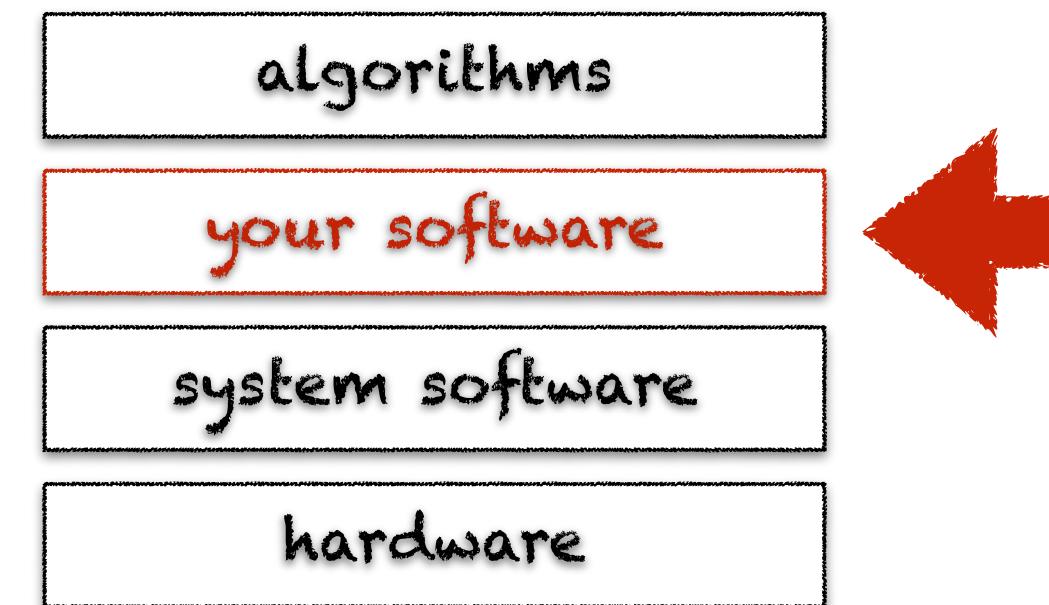




classes,
objects &
methods

learning objectives



- learn about encapsulation and abstraction
- learn about classes, objects and methods
- learn how to create your own classes
- learn about modularization and code reuse

software engineering

an algorithm focuses on a specific computational procedure that solve a particular problem

a complete program is however composed of many such algorithms, resulting in many lines of code

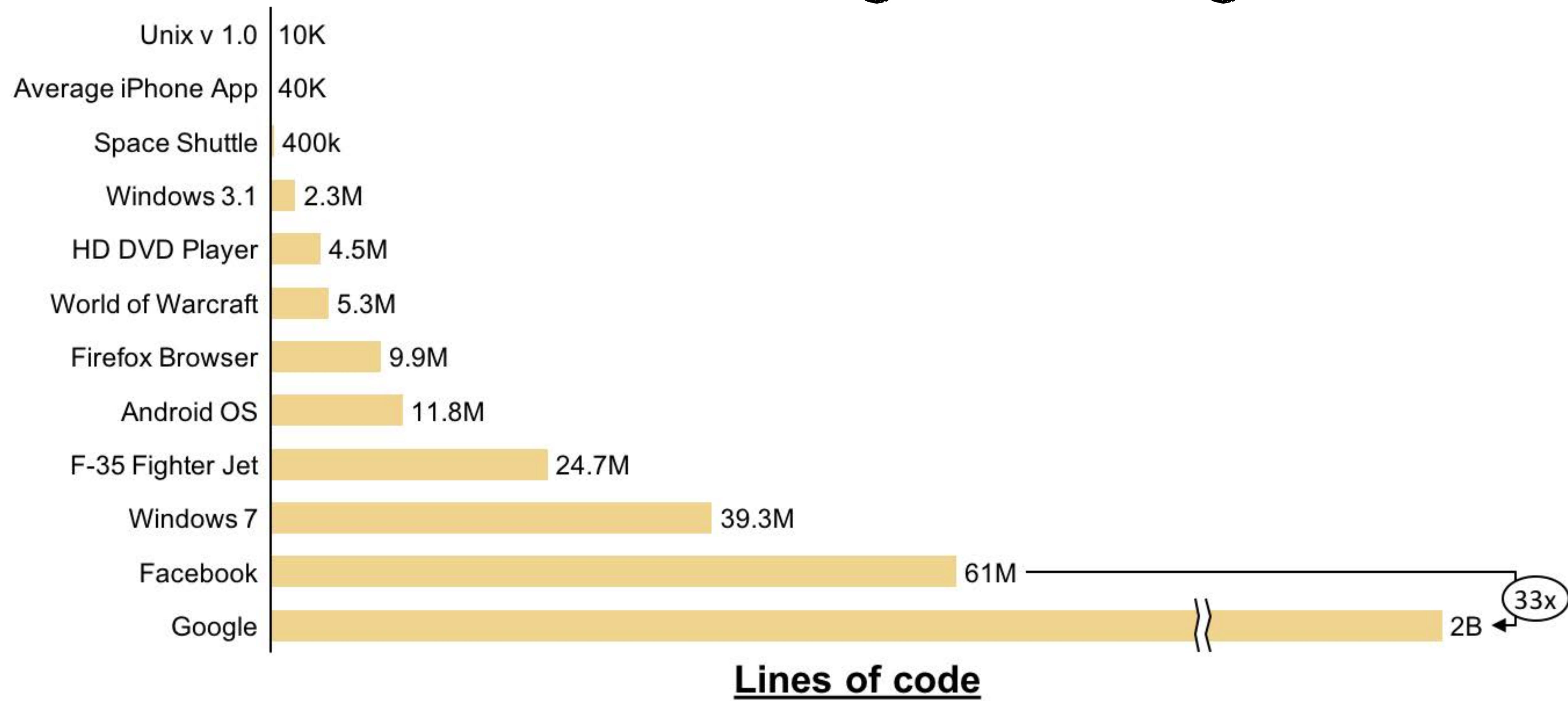
the linux kernel consists of 15+ million lines of code*

the google codebase consists of 2+ billion lines of code*

*January 2018

we need software engineering tools
to manage this complexity

software engineering



software engineering

software engineering tools are of different kinds, e.g., methodologies (agile), abstract notations (uml), source-oriented tools (ide, git), programming language constructs that help encapsulate complexity, e.g., objects, functions, etc.

in this course, we are mainly interested in programming language constructs, in particular objects and functions

today we focus on classes, objects and methods

this is known as the object-oriented approach

what's an objects?

represents particular
“things” from the real
world, or from some
problem domain (e.g., “my
blue rocking chair”)



what's a class?

represents
all objects of
a given kind,
e.g., "chairs"



specification vs implementation

what it does



how it does it

specification viewpoint

the viewpoint of someone
simply wanting to use
objects (not design them)

no need to know how
objects are built to use
them, only **what can be
done with them**



encapsulation principle: allows
us to hide (encapsulate) the
complexity of objects

a **class specifies** the set of
common behaviors offered by
objects (instances) of that class

methods & parameters



`chair.rotate(45)`

object have methods
(operations) that can be
invoked (called) and define
their possible behaviors

when we want an object to
do something for us, we call
one of its methods

the set of (public) methods of an object can be seen
as its contract with the world (its specification)

methods & parameters



chair.rotate(45)

methods may have
parameters to pass
additional information
needed to execute it

implementation viewpoint



how it does it

the implementation viewpoint is concerned with how an object actually fulfills its specification (its contract)

the fields and methods define how the object will behave and are defined by its class

class

instances

many instances
(objects) can be
created from a

single class

the class can be seen
as a kind of object
factory (or a mold)



fields

the source code of **classes** defines
the attributes (**fields**) and methods
all objects of the class have

class Chair	
color	(string)
model	(string)
isBroken	(boolean)
age	(integer)

field values

each object stores its own
values for each field

field values
represent the
object's state



instance myChair

color

"green"

model

"shell"

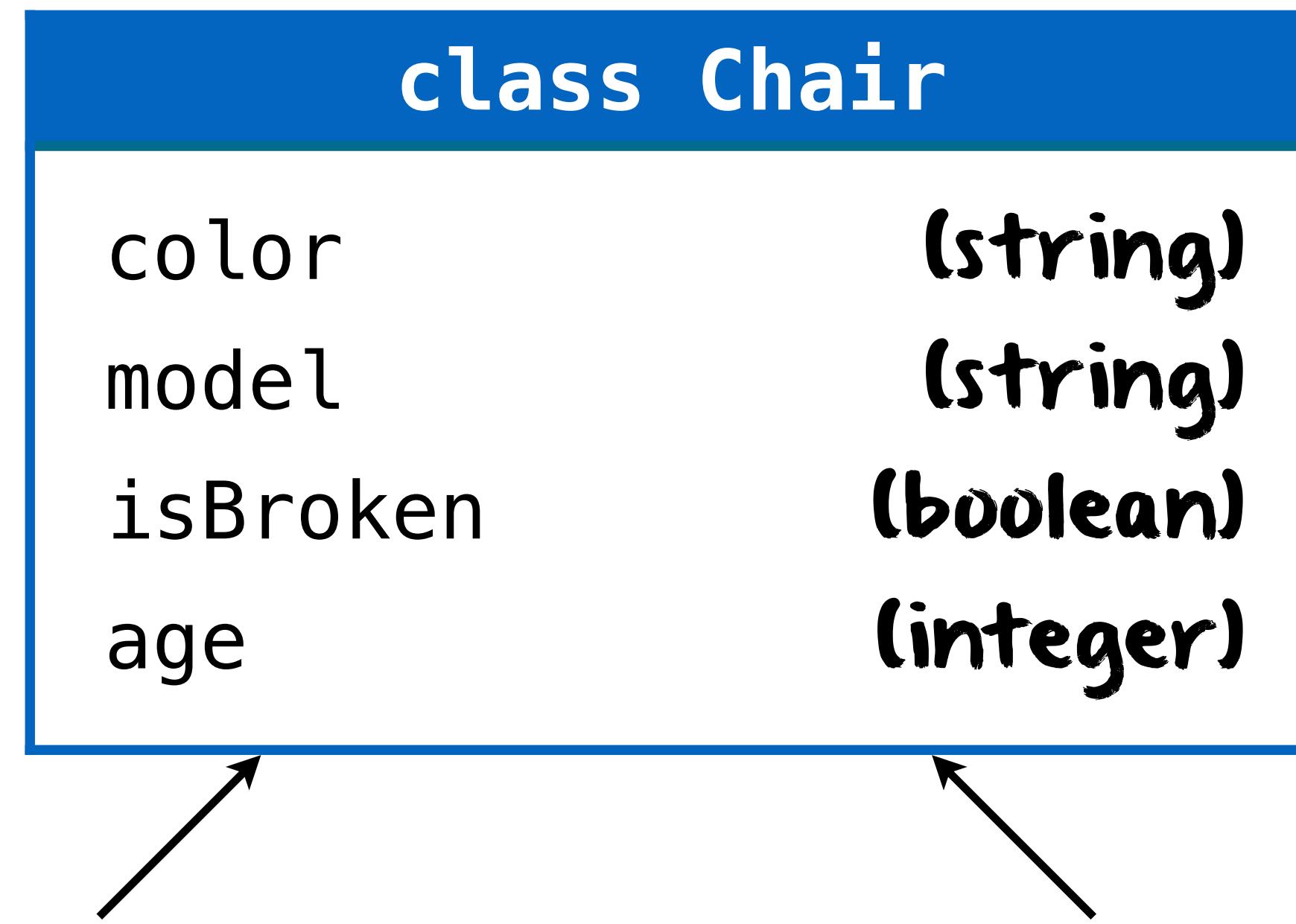
isBroken

false

age

5

two chair instances



instance myFirstChair

color	"brown"
model	"wood"
isBroken	false
age	50

instance mySecondChair

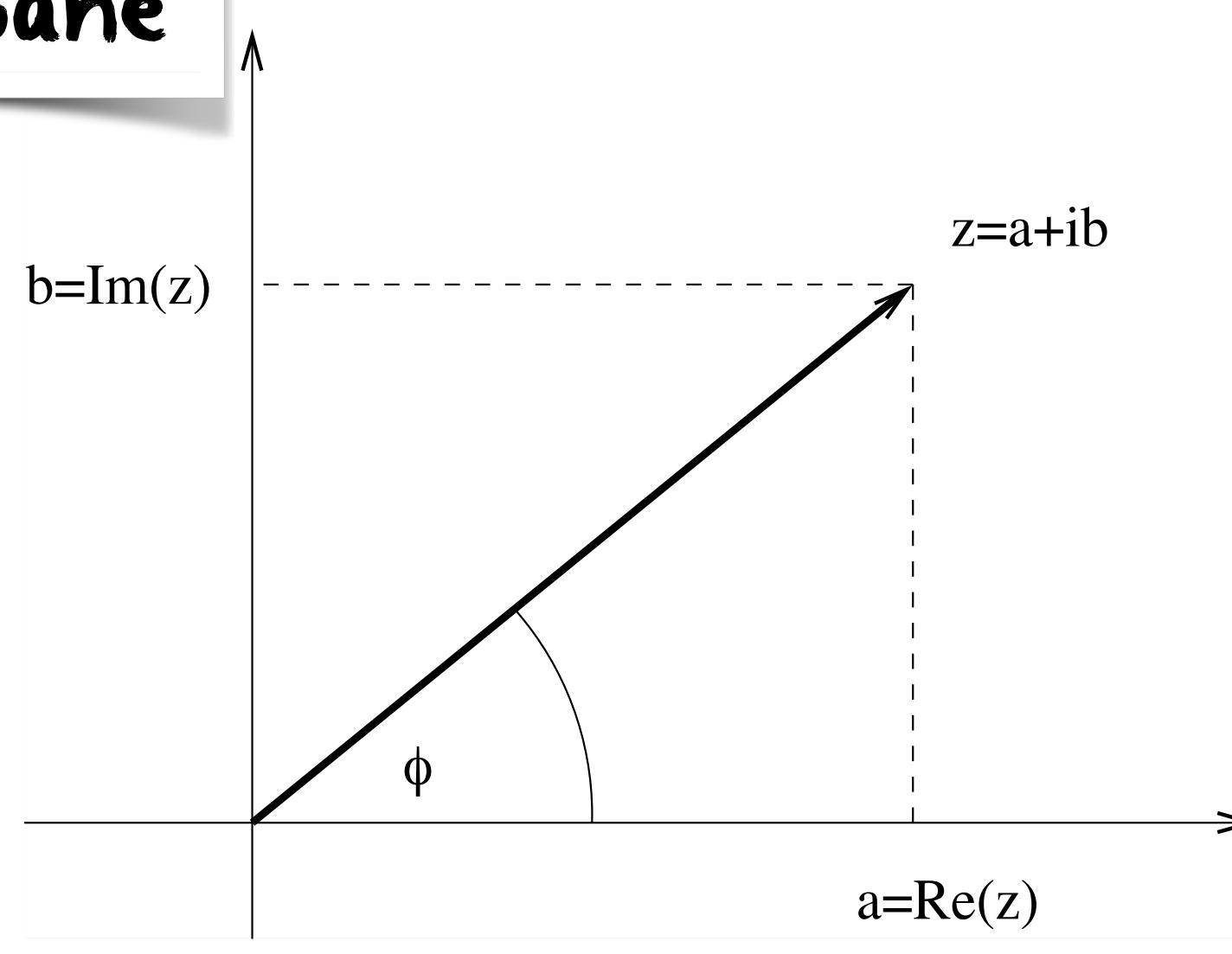
color	"green"
model	"shell"
isBroken	false
age	5

complex numbers quick reminder

$$z = a + ib$$

with $i = \sqrt{-1}$
 $a = \text{Re}(z)$
 $b = \text{Im}(z)$

complex plane



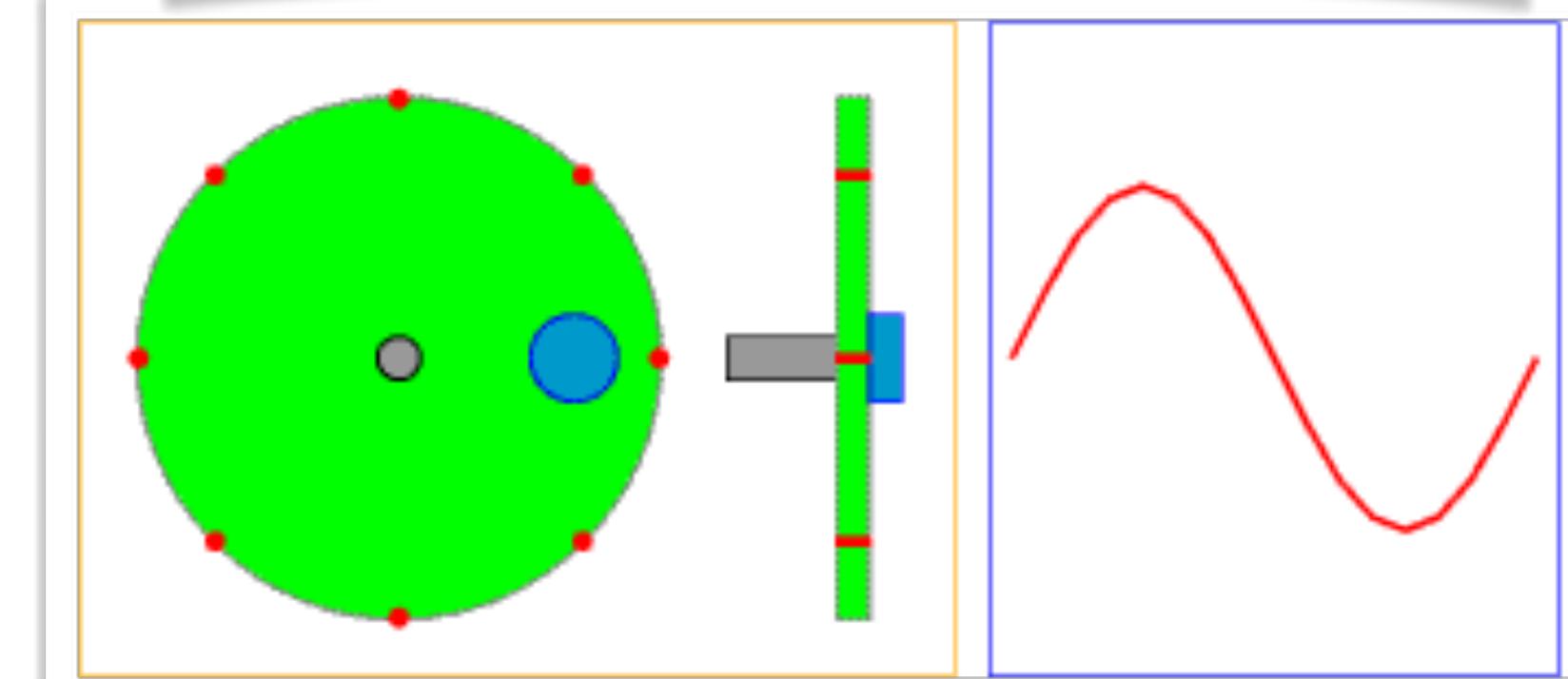
$$\tan \phi = \frac{\text{Im}(z)}{\text{Re}(z)}$$

$$z = |z|(\cos \phi + i \sin \phi)$$

$$e^{i\phi} = \cos \phi + i \sin \phi$$

$$z = |z|e^{i\phi}$$

intuitive interpretation



complex numbers quick reminder

addition $(a + bi) + (c + di) = (a + c) + (b + d)i$

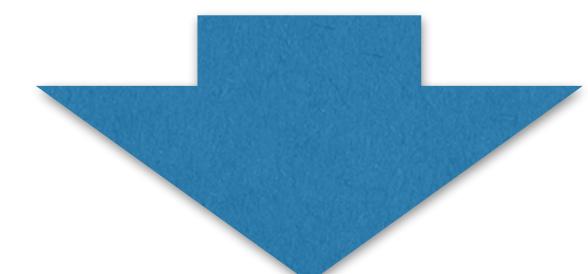
subtraction $(a + bi) - (c + di) = (a - c) + (b - d)i$

multiplication $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

```
public class Complex {  
    public final double re;  
    public final double im;  
  
    public Complex(double re, double im) { this.re = re; this.im = im; }  
  
    public Complex add(Complex other) {  
        return new Complex(this.re + other.re, this.im + other.im);  
    }  
  
    public Complex subtract(Complex other) {  
        return new Complex(this.re - other.re, this.im - other.im);  
    }  
  
    public Complex multiply(Complex other) {  
        return new Complex(this.re * other.re - this.im * other.im,  
                           this.im * other.re + this.re * other.im);  
    }  
}
```



```
Complex z1 = new Complex(2, -1);  
Complex z2 = new Complex(2, -4);  
  
Complex z = z1.add(z2);  
System.out.println(z1 + " + " + z2 + " = " + z);  
  
z = z1.subtract(z2);  
System.out.println(z1 + " - " + z2 + " = " + z);  
  
z = z1.multiply(z2);  
System.out.println(z1 + " * " + z2 + " = " + z);
```



```
2-i + 2-4i = 4-5i  
2-i - 2-4i = 3i  
2-i * 2-4i = -10i
```

complex numbers quick reminder

addition $(a + bi) + (c + di) = (a + c) + (b + d)i$

subtraction $(a + bi) - (c + di) = (a - c) + (b - d)i$

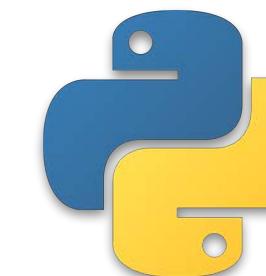
multiplication $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

```
class Complex(object):
    def __init__(self, re, im):
        self.re = re
        self.im = im

    def add(self, other):
        return Complex(self.re + other.re,
                      self.im + other.im)

    def sub(self, other):
        return Complex(self.re - other.re,
                      self.im - other.im)

    def mul(self, other):
        return Complex(self.re * other.re - self.im * other.im,
                      self.im * other.re + self.re * other.im)
```

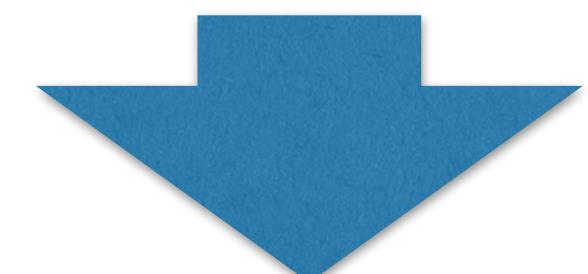


```
z1 = Complex(2,-1)
z2 = Complex(2,-4)

z = z1.add(z2)
print("{0} + {1} = {2}".format(z1,z2,z))

z = z1.sub(z2)
print("{0} - {1} = {2}".format(z1,z2,z))

z = z1.mul(z2)
print("{0} * {1} = {2}".format(z1,z2,z))
```



```
2-i + 2-4i = 4-5i
2-i - 2-4i = 3i
2-i * 2-4i = -10i
```

complex numbers quick reminder

addition $(a + bi) + (c + di) = (a + c) + (b + d)i$

subtraction $(a + bi) - (c + di) = (a - c) + (b - d)i$

multiplication $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

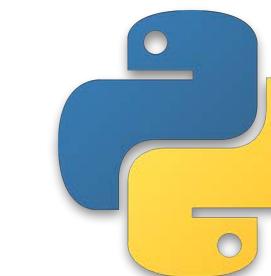
operator overloading

```
class Complex(object):
    def __init__(self, re, im):
        self.re = re
        self.im = im

    def __add__(self, other):
        return Complex(self.re + other.re,
                      self.im + other.im)

    def __sub__(self, other):
        return Complex(self.re - other.re,
                      self.im - other.im)

    def __mul__(self, other):
        return Complex(self.re * other.re - self.im * other.im,
                      self.im * other.re + self.re * other.im)
```

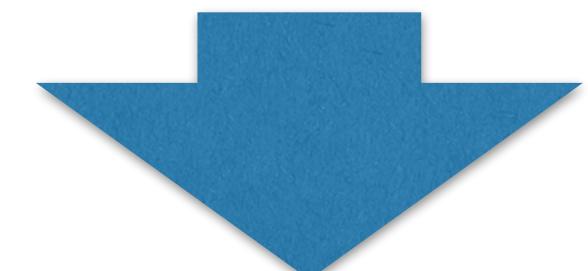


```
z1 = Complex(2,-1)
z2 = Complex(2,-4)

z = z1 + z2
print("{0} + {1} = {2}".format(z1,z2,z))

z = z1 - z2
print("{0} - {1} = {2}".format(z1,z2,z))

z = z1 * z2
print("{0} * {1} = {2}".format(z1,z2,z))
```



```
2-i + 2-4i = 4-5i
2-i - 2-4i = 3i
2-i * 2-4i = -10i
```

complex numbers

no operator overloading in java

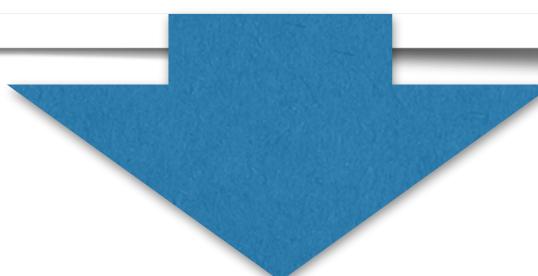
BUT

method overloading: yes

```
public class Complex {  
    public final double re;  
    public final double im;  
    :  
    public Complex add(Complex other) {  
        return new Complex(this.re + other.re, this.im + other.im);  
    }  
    :  
    public Complex add(double other) {  
        return new Complex(this.re + other, this.im);  
    }  
    :  
}
```



```
Complex z1 = new Complex(2, -1);  
Complex z2 = new Complex(2, -4);  
  
Complex z = z1.add(z2);  
System.out.println(z1 + " + " + z2 + " = " + z);  
  
double d = 10;  
z = z1.add(10);  
System.out.println(z1 + " + " + d + " = " + z);
```



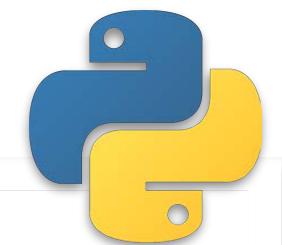
$2-i + 2-4i = 4-5i$
 $2-i + 10 = 12-10i$

class declaration

```
class declaration    public class Complex {  
field declaration      public final double re;  
constructor            public final double im;  
constructor            public Complex(double re, double im) { this.re = re; this.im = im; }  
method                public Complex add(Complex other) {  
method overloading      return new Complex(this.re + other.re, this.im + other.im);  
                         }  
                         :  
method overloading      public Complex add(double other) {  
method overloading      return new Complex(this.re + other, this.im);  
                         }  
                         }
```

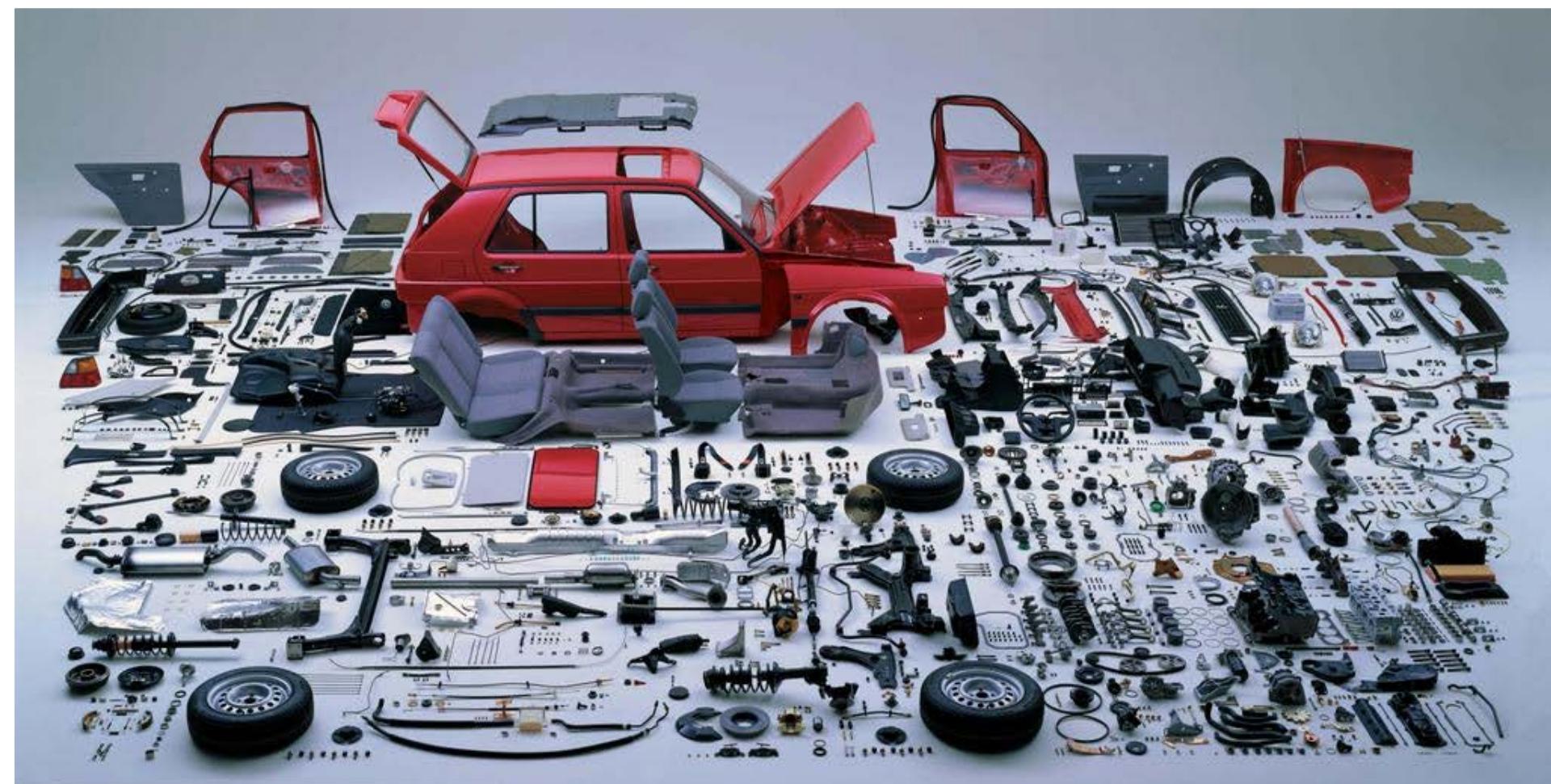


```
class declaration        class Complex(object):  
constructor              def __init__(self, re, im):  
field declaration          self.re = re  
method                  self.im = im  
operator overloading      def add(self, other):  
                           return Complex(self.re + other.re,  
                           self.im + other.im)  
operator overloading      def __add__(self, other):  
                           return Complex(self.re + other.re,  
                           self.im + other.im)
```



abstraction & modularization

modularization consists in dividing a complex object into elemental objects that can be developed independently



the encapsulation offered by objects is the cornerstone of modularization because it hides implementation details

once elemental objects have been developed and tested, they can be assembled into a more complex object

this is known as code reuse

abstraction & modularization

example of a digital clock

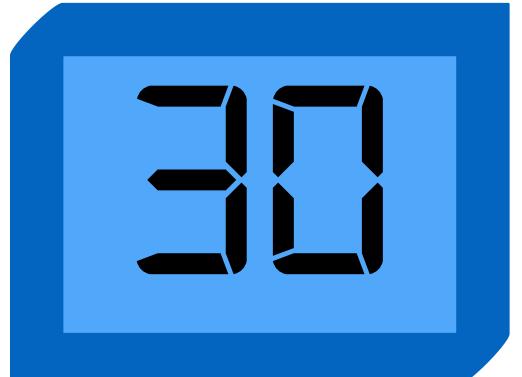


one four-digit display?



OR

two two-digit displays?



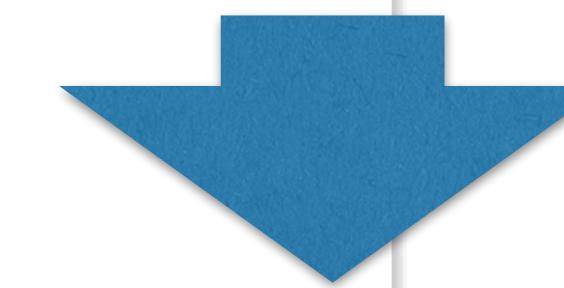


NumberDisplay class



```
public class NumberDisplay {  
    private int limit;  
    private int value;  
  
    public NumberDisplay(int limit, int value){  
        this.limit = limit;  
        this.value = value;  
    }  
    public NumberDisplay(int limit){  
        this(limit, 0);  
    }  
    public int get() { return value; }  
    public void set(int value) {  
        this.value = value % limit;  
    }  
    public void increment(){  
        value = (value + 1) % limit;  
    }  
    public String toString(){  
        if(value < 10) { return "0" + String.valueOf(value); }  
        else { return String.valueOf(value); }  
    }  
}
```

```
NumberDisplay number = new NumberDisplay(24);  
System.out.println("number = " + number);  
  
number.set(22);  
System.out.println("number = " + number);  
  
number.increment();  
System.out.println("number = " + number);  
  
number.increment();  
System.out.println("number = " + number);
```



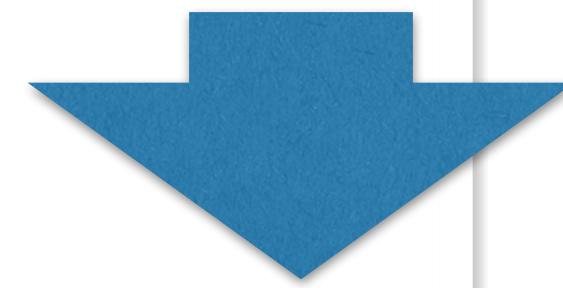
```
number = 00  
number = 22  
number = 23  
number = 00
```



ClockDisplay class

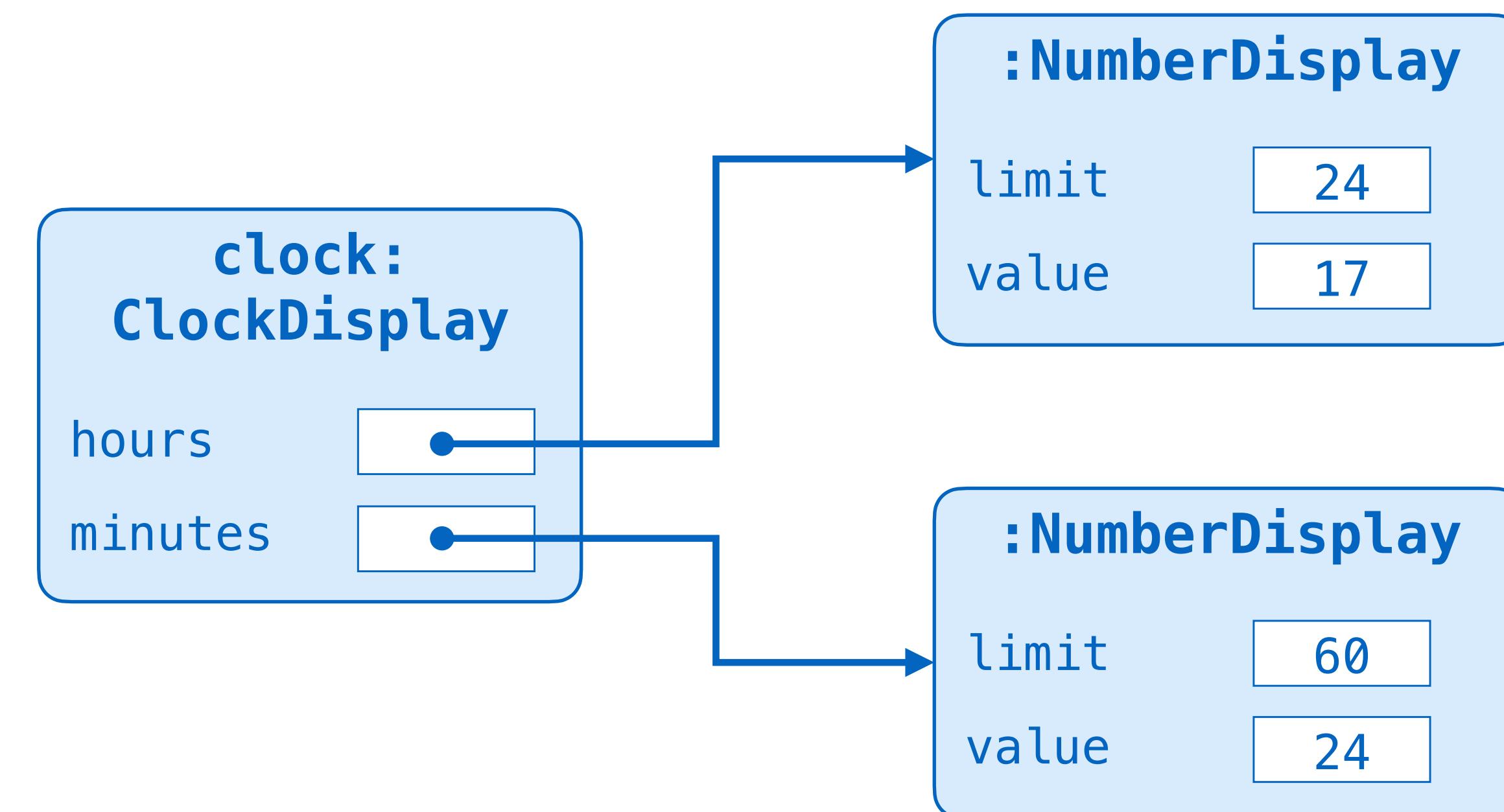
```
public class ClockDisplay {  
    private NumberDisplay hours;  
    private NumberDisplay minutes;  
    private String displayString;  
  
    public ClockDisplay(){  
        hours = new NumberDisplay(24,0);  
        minutes = new NumberDisplay(60, 0);  
    }  
    public void timeTick(){  
        minutes.increment();  
        if(minutes.get() == 0) {  
            hours.increment();  
        }  
    }  
    public void set(int hours, int minutes) {  
        this.hours.set(hours);  
        this.minutes.set(minutes);  
    }  
    public String toString(){  
        return hours + ":" + minutes;  
    }  
}
```

```
ClockDisplay clock = new ClockDisplay();  
System.out.println("clock = " + clock);  
  
clock.set(10,58);  
System.out.println("clock = " + clock);  
  
clock.timeTick();  
System.out.println("clock = " + clock);  
  
clock.timeTick();  
System.out.println("clock = " + clock);  
  
clock.set(23,59);  
System.out.println("clock = " + clock);  
  
clock.timeTick();  
System.out.println("clock = " + clock);
```

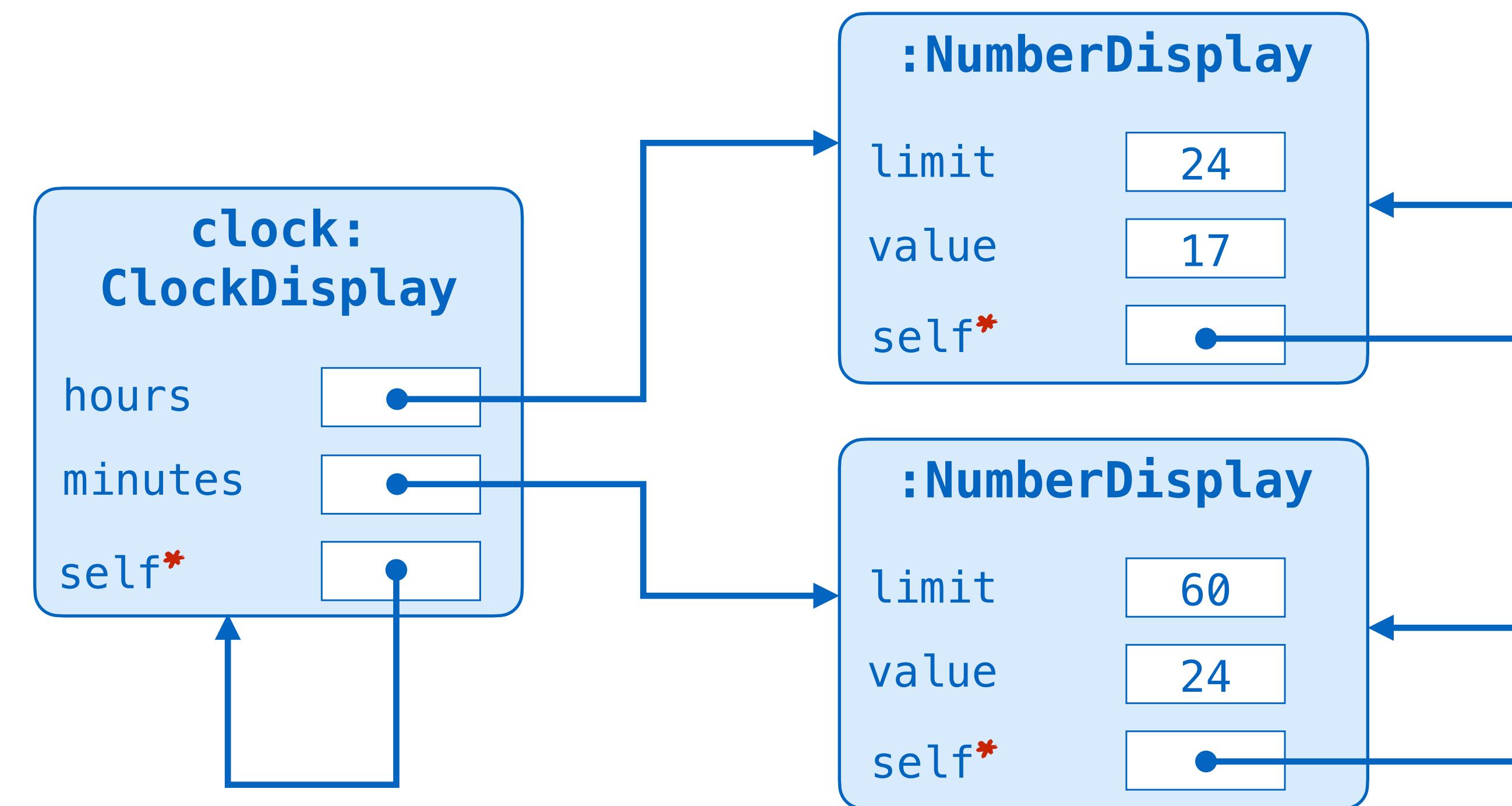


```
clock = 00:00  
clock = 10:58  
clock = 10:59  
clock = 11:00  
clock = 23:59  
clock = 00:00
```

object diagram



object diagram



*or this in some languages