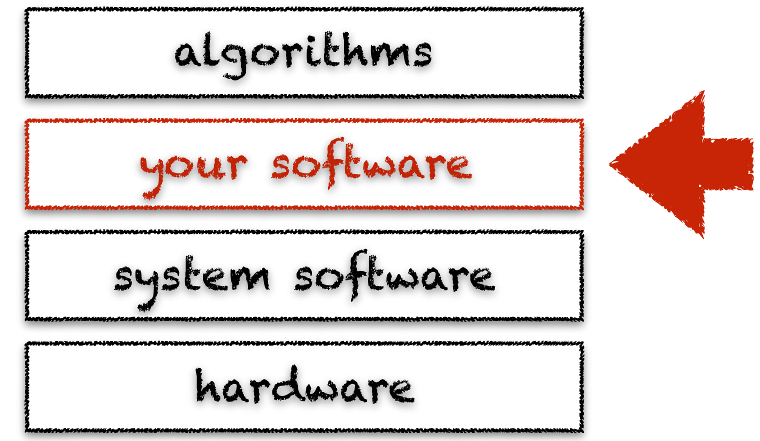
classes, objects & methods

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*

*September 2015

we need software engineering tools to manage this complexity

software engineering

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

in this course, we are only 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

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

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

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

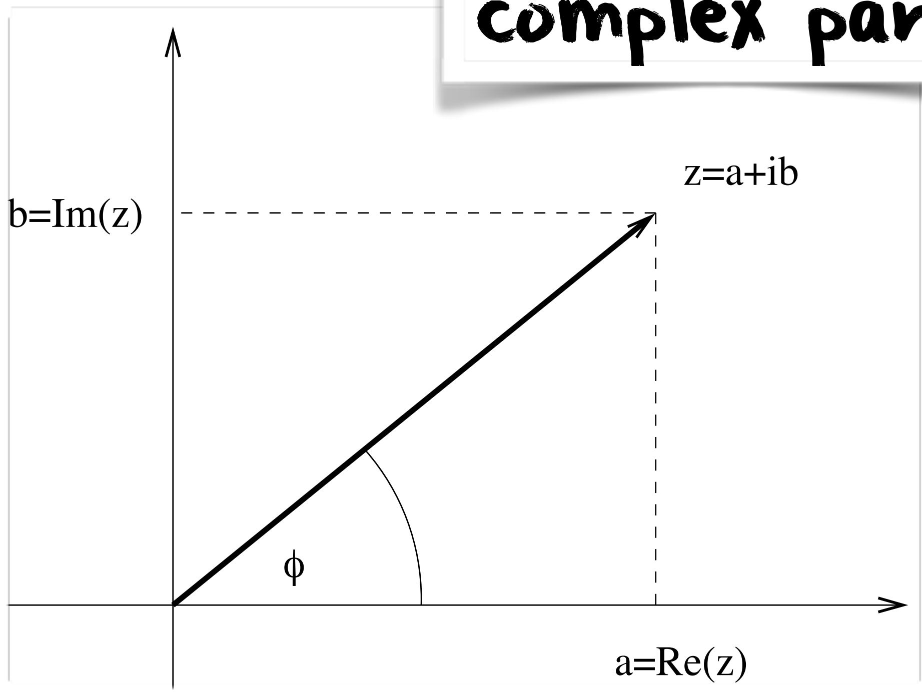
instance mySecondChair	
color	"green"
model	"shell"
isBroken	false
age	5

complex numbers quick reminder

$$z = a + ib$$

$$\begin{aligned} \textit{with} \quad i &= \sqrt{-1} \\ a &= \mathrm{Re}\,(z) \\ b &= \mathrm{Im}\,(z) \end{aligned}$$

complex pane



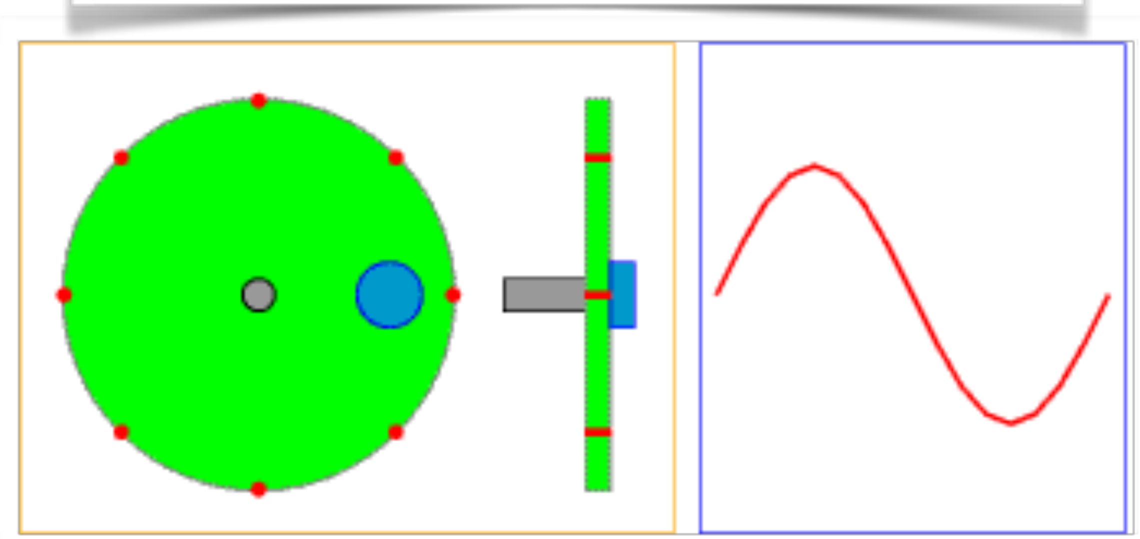
$$\tan\phi = \frac{\mathrm{Im}\,(z)}{\mathrm{Re}\,(z)}$$

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

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

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

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

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

```
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
```

operator overloading

complex numbers

operator & constructor overloading

```scala
class Complex(val re: Double, val im: Double) {

  def add(c: Complex) = new Complex(re + c.re, im + c.im)

  def +(c: Complex) = new Complex(re + c.re, im + c.im)

  def +(d: Double) = new Complex(re + d, im)

  def this(re: Double) = this(re, 0)
}
implicit def fromDouble(d: Double) = new Complex(d)
```

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

var z = z1.add(z2)
println(s"$z1 + $z2 = $z")
z = z1 + z2
println(s"$z1 + $z2 = $z")
z = z1 + 6
println(s"$z1 + 6 = $z")

z = 6 + z1
println(s"6 + $z1 = $z")
```

implicit conversion

```
2.0-1.0i + 2.0-4.0i = 4.0-5.0i
2.0-1.0i + 2.0-4.0i = 4.0-5.0i
2.0-1.0i + 6 = 8.0-1.0i
6 + 2.0-1.0i = 8.0-1.0i
```

class declaration

```scala
class Complex(val re: Double, val im: Double) {
  def add(c: Complex) = new Complex(re + c.re, im + c.im)
  def +(c: Complex) = new Complex(re + c.re, im + c.im)
  def +(d: Double) = new Complex(re + d, im)
  def this(re: Double) = this(re, 0)
}
implicit def fromDouble(d: Double) = new Complex(d)
```

class declaration & constructor

method

operator overloading

constructor overloading

implicit conversion

class declaration

constructor

method

operator overloading

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

abstraction & modularization

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

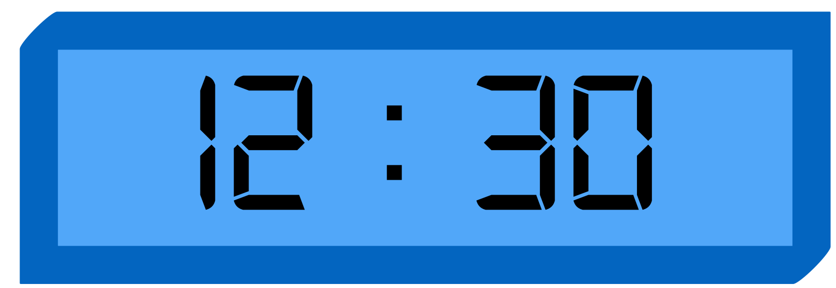
once elemental objects have been developed and tested, they can be assembled into a more complex object

this is known as code reuse

abstraction & modularization

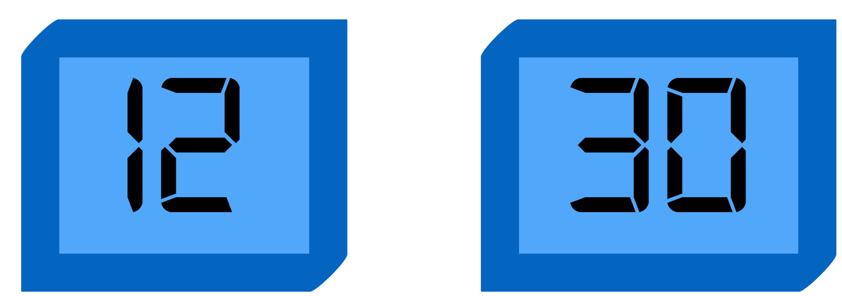
example of a digital clock

one four-digit display?



OR

two two-digit displays?



NumberDisplay class


```scala
class NumberDisplay(val limit: Int, private var value : Int = 0) {

  def increment() {
    value = (value + 1) % limit
  }

  def set(value: Int) {
    this.value = value % limit
  }

  def get : Int = { this.value }

  override def toString: String = {
    if(value < 10)
      "0" + value
    else
      value.toString
  }
}
```

```scala
val number = new NumberDisplay(24)
println(s"number = $number")

number.set(22)
println(s"number = $number")

number.increment()
println(s"number = $number")

number.increment()
println(s"number = $number")
```

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

ClockDisplay class

```scala
class ClockDisplay() {
  val hours = new NumberDisplay(24)
  val minutes = new NumberDisplay(60)

  def timeClick {
    minutes.increment()
    if (minutes.get == 0)
      hours.increment()
  }

  def set(hours:Int, minutes:Int) {
    this.hours.set(hours)
    this.minutes.set(minutes)
  }

  override def toString: String =  hours + ":" + minutes
}
```

```scala
val clock = new ClockDisplay
println(s"clock = $clock")

clock.set(10,58)
println(s"clock = $clock")

clock.timeClick
println(s"clock = $clock")

clock.timeClick
println(s"clock = $clock")

clock.set(23,59)
println(s"clock = $clock")

clock.timeClick
println(s"clock = $clock")
```

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

object diagram