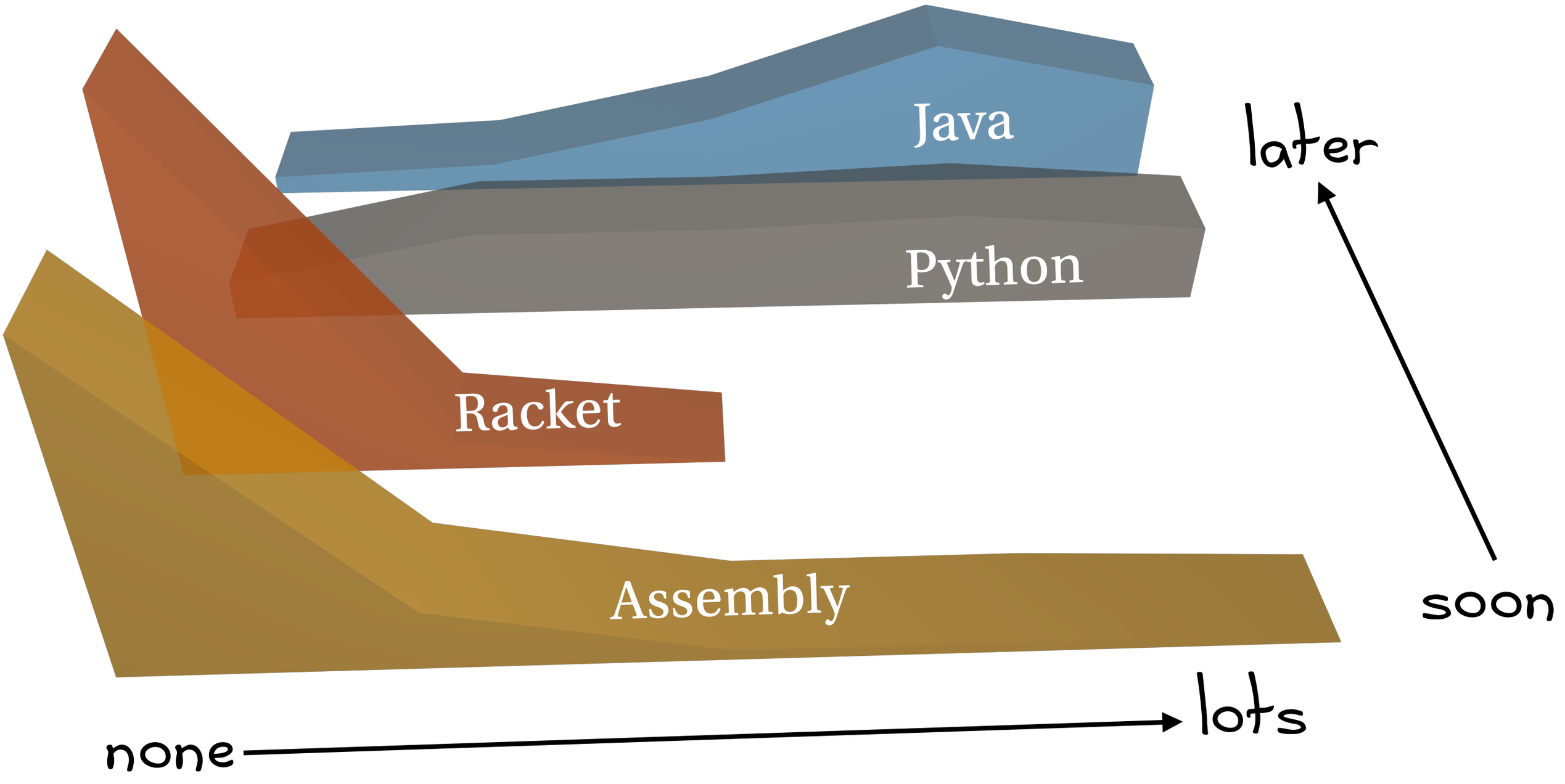
OOP in Java

Prior experience: programming languages



Java is a byte-compiled language.

Java has
static types.

primitive values *vs* objects

Memory model

Primitive values

`int` • `double` • `boolean`
other built-in types …

```
int x = 3;
int y = x;
int z = 3;
```

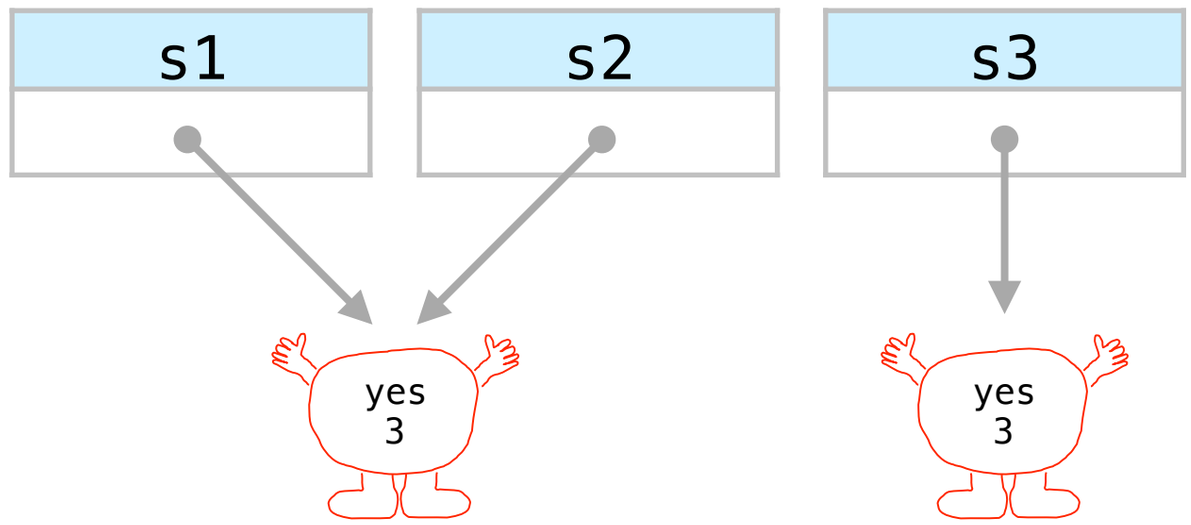
x	y	z
3	3	3

Java directly stores primitive values.

Objects

`String` • `LinkedList`
other library & user-defined types …

```
String s1 = "yes";
String s2 = s1;
String s3 = "yes";
```



Java stores references to objects.

== *vs* .equals

==

compares what's in the box

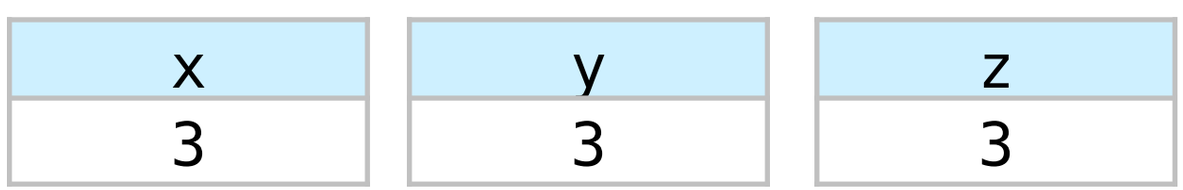
```
x == y;   // true
y == z;   // true
s1 == s2; // true
s2 == s3; // false
```

.equals

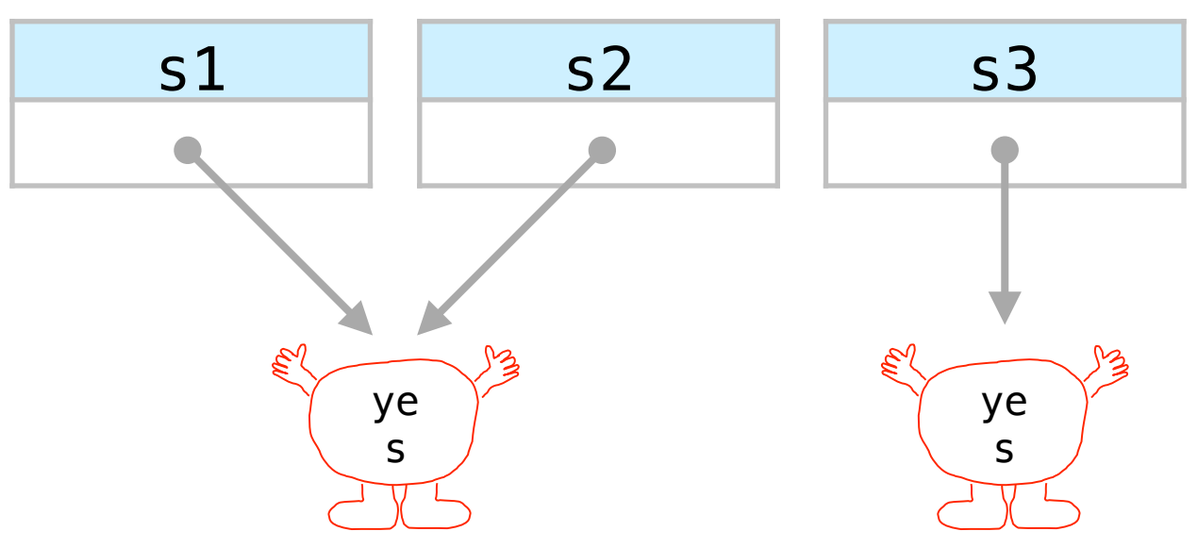
calls a method (usually checks for equal values)

```
s1.equals(s2);   // true
s2.equals(s3);   // true
```

```
int x = 3;
int y = x;
int z = 3;
```




```
String s1 = "yes";
String s2 = s1;
String s3 = "yes";
```



	primitives	**objects**
variable stores the value	✔	
variable stores a reference		✔
supports ==	✔	✔ but it 's probably not what you want
supports `.equals`		✔
we can define new kinds		✔
type name starts with lower-case letter	✔ `int value  //`	
type name starts with upper-case letter		✔ `Dog lucky  // object`

Object-oriented Programming (again ☺)

What is object-oriented programming good for?

Object-oriented programming helps us manage the complexity of programs by:

1. **combining data with the behavior** that operates over it
2. breaking large programs into smaller, **self-contained** pieces
3. separating **interface** (*what* a piece of code can do) from **implementation** (*how* that piece of code works)

Note: there's an underlying assumption that your program is complex enough to need OOP.

programs
=
data + behavior

encapsulation
(separate interface
from implementation)

object

- combines data & behavior
- access only through interface
- knows about itself
  (can access its own data and behavior)

an object is sort-of
like a little state machine!

Object-oriented programming languages **differ** in:

- how the programmer specifies an object's **interface**
- how the programmer specifies an object's **implementation**
- how objects are **created**, **initialized**, **queried**, and **updated**
- **encapsulation** mechanism
  how strictly the language *enforces* the separation between interface & implementation

Object-oriented Programming

in Java

A class is like…

a cookie cutter

ecx.images-amazon.com/images/I/21owTyO6HaL.jpg

Objects are like…

cookies

eclecticrecipes.com/wp-content/uploads/2013/02/heart-6.jpg

images.edge-generalmills.com/9b6a8635-686e-4b7d-863b-7dd3d8d25a04.jpg

A class is like…

factory

si.wsj.net/public/resources/images/P1-AO506_TURNPI_G_20090129173936.jpg

Objects are like…

cars

A class is like…

factory

www.tipsquirrel.com/wp-content/uploads/2010/09/Extrude1.jpg

Objects are like…

delicious,
totally edible
playdough

class: a blueprint for an object; contains implementation

object: a self-contained instance of a class

field: stores data

method: defines a behavior

constructor: initializes an object's fields

getter: a method that lets us read an object's data

setter: a method that lets us change an object's data

this: how an object knows about itself

interface: what an object can do

implementation: how an object does its thing

public: indicates a piece of the interface

private: indicates a piece of the implementation

```java
class Point {
    /** the x (horizontal) coordinate */
    private double x;

    /** the y (vertical) coordinate */
    private double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return this.x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return this.y;
    }

    public void setY(double y) {
        this.y = y;
    }

    /**
     * returns the sum of this point and another
     *
     * @param other another Point object
     * @return a new Point, the sum of this and other
     */
    public Point add(Point other) {
      return new Point(this.getX() + other.getX(),
                       this.getY() + other.getY());
    }
}
```

field definition

Javadoc comment

constructor definition

getter

field access

setter

method definition

getter

setter

(this is an object)

class

Javadoc comment

constructor call

lots of (getter) method calls!

object!

Be on the lookout for

- Where's the interface? Where's the implementation?
- How to create, initialize, query, and update an object
- How does Java enforce separation of interface & implementation?

- object-oriented vocabulary
- good programming practices
- good programming style
- when (not) to use a particular object-oriented feature

- how to do things in Java
- how to do things in Eclipse

- questions / confusions / pondering

An Excel-ent analogy

Fields are like a spreadsheet

Class definition ≈ columns

a class defines the names and types (but not the values) of fields

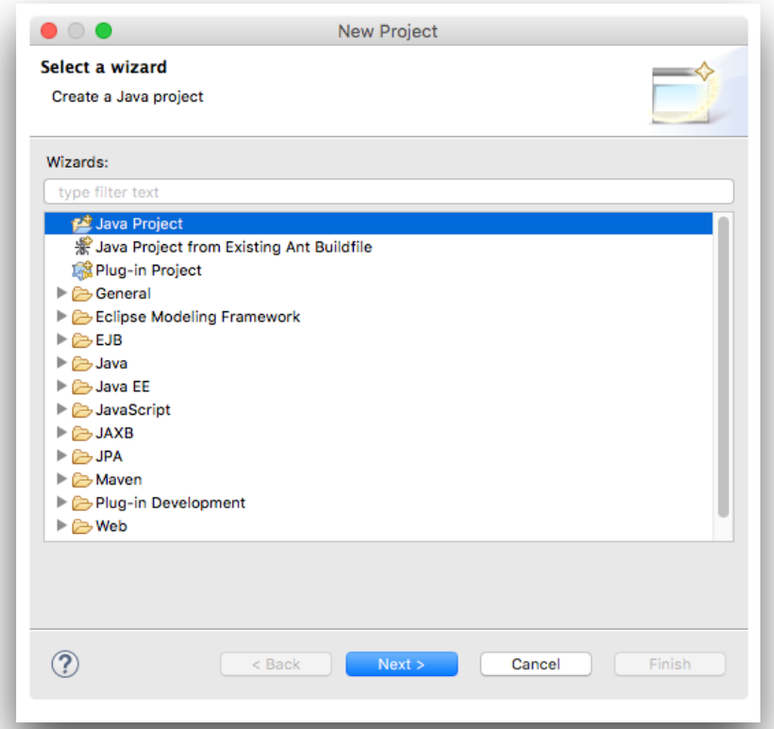
	color	capacity	fullness
Colleen's mug	blue	100	100
Ben's jug	puce	1000	500
Zach's coffee cup	white & green	100000	0

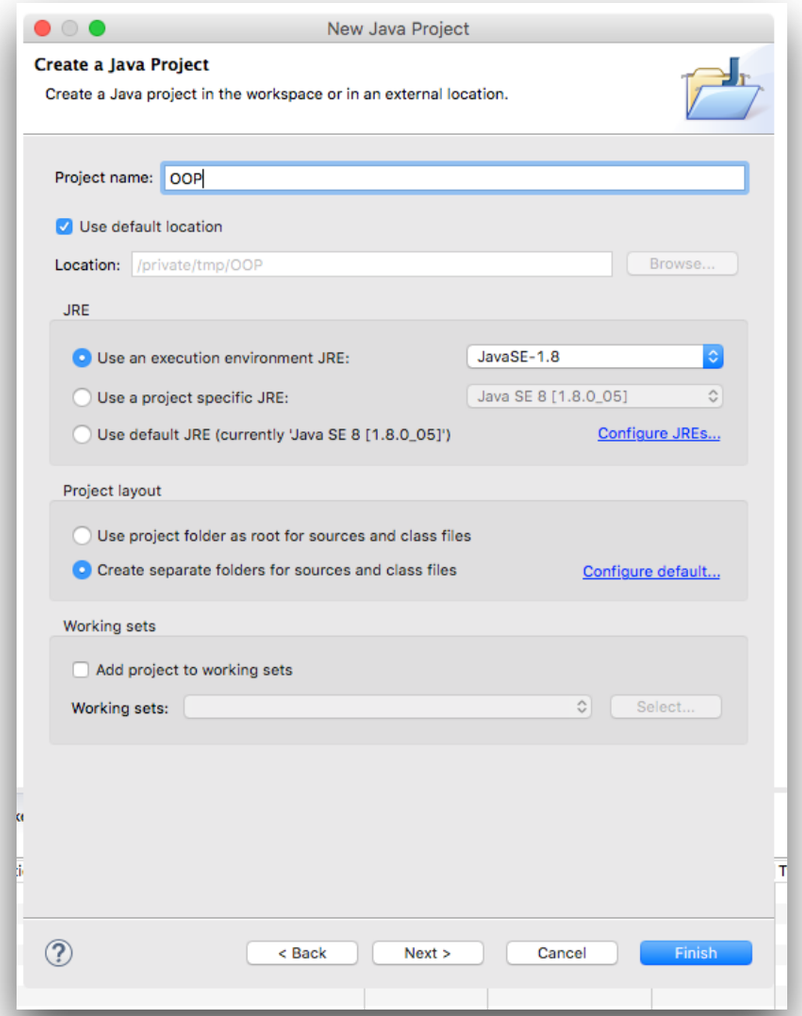
Objects ≈ rows

each object has specific values for its field

How to create an Eclipse Project

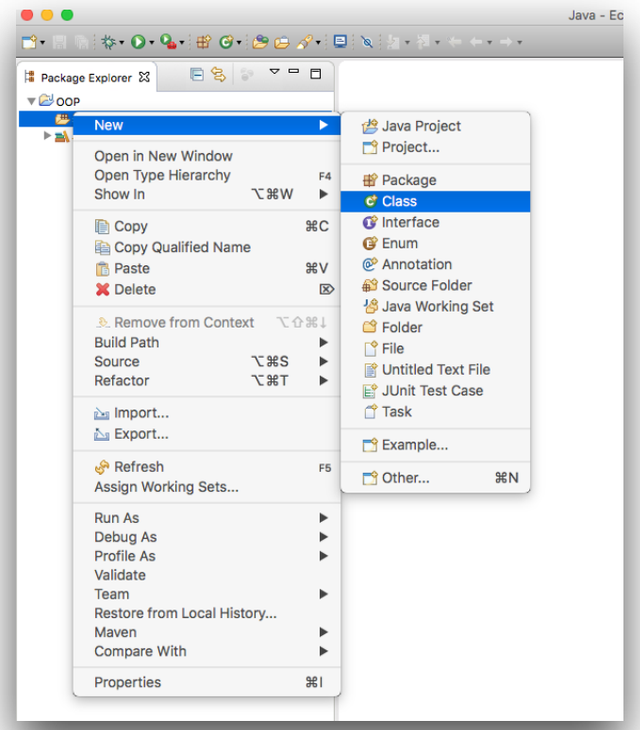
File → New Java Project



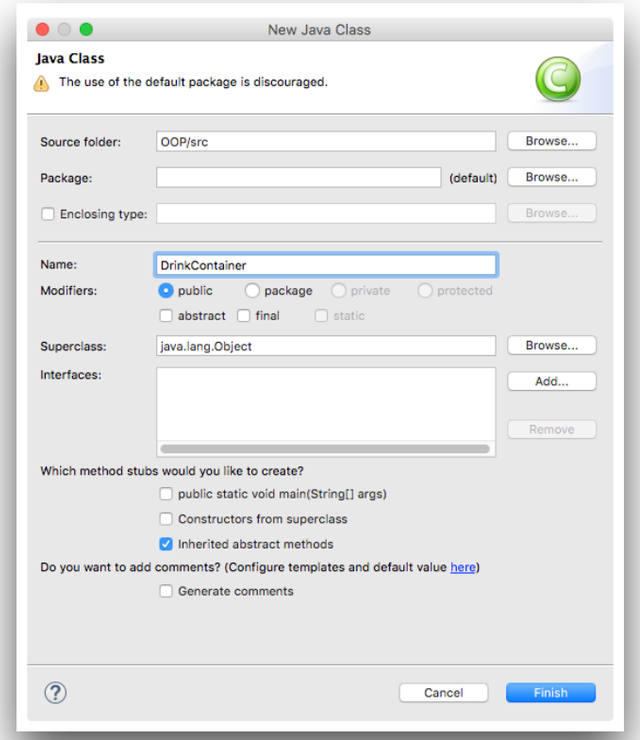


How to create a new Java class

Right-click the src folder

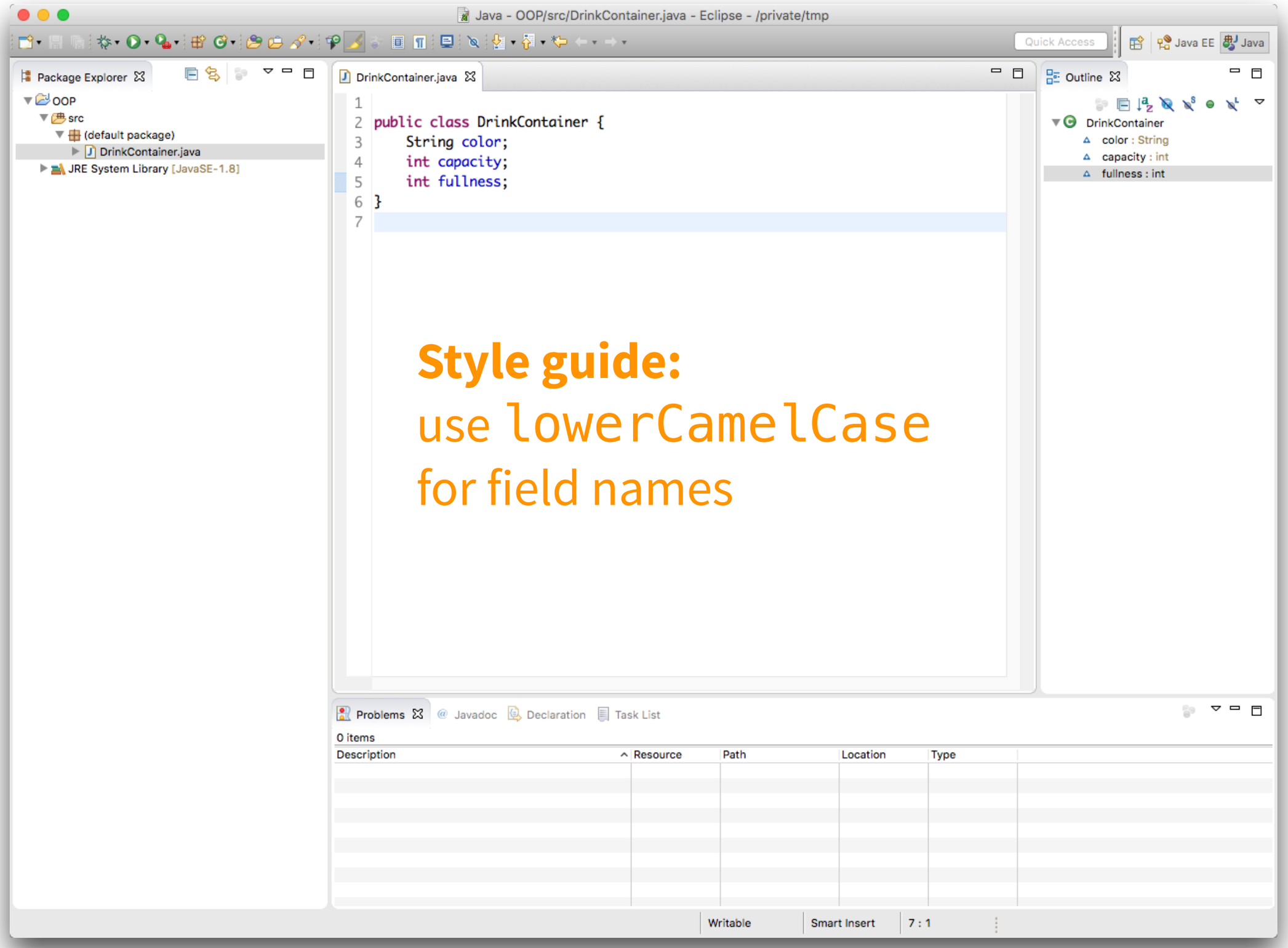


Give the class a good name



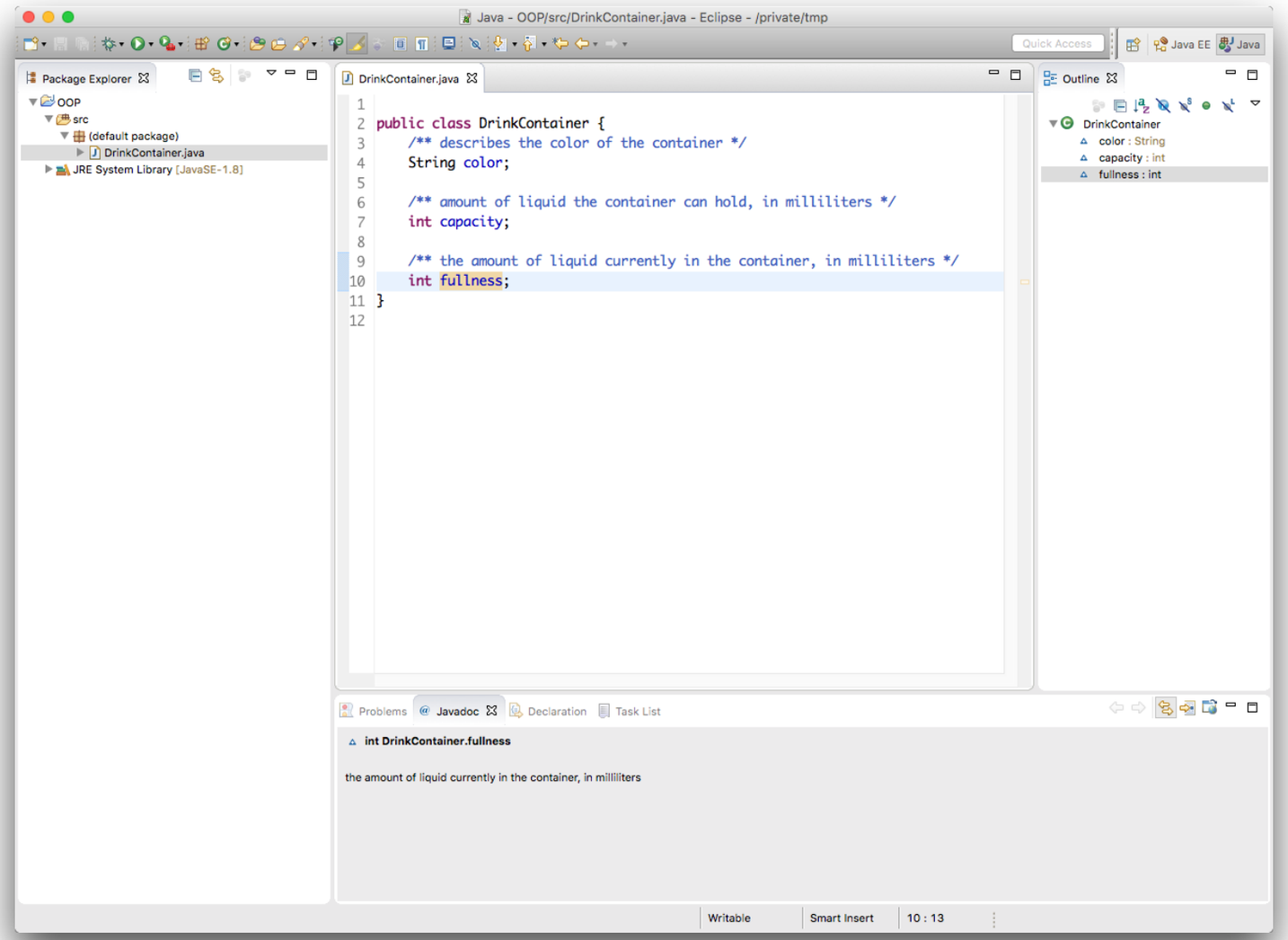
Style guide:
use `UpperCamelCase`
for class names

Field definitions go at top of class

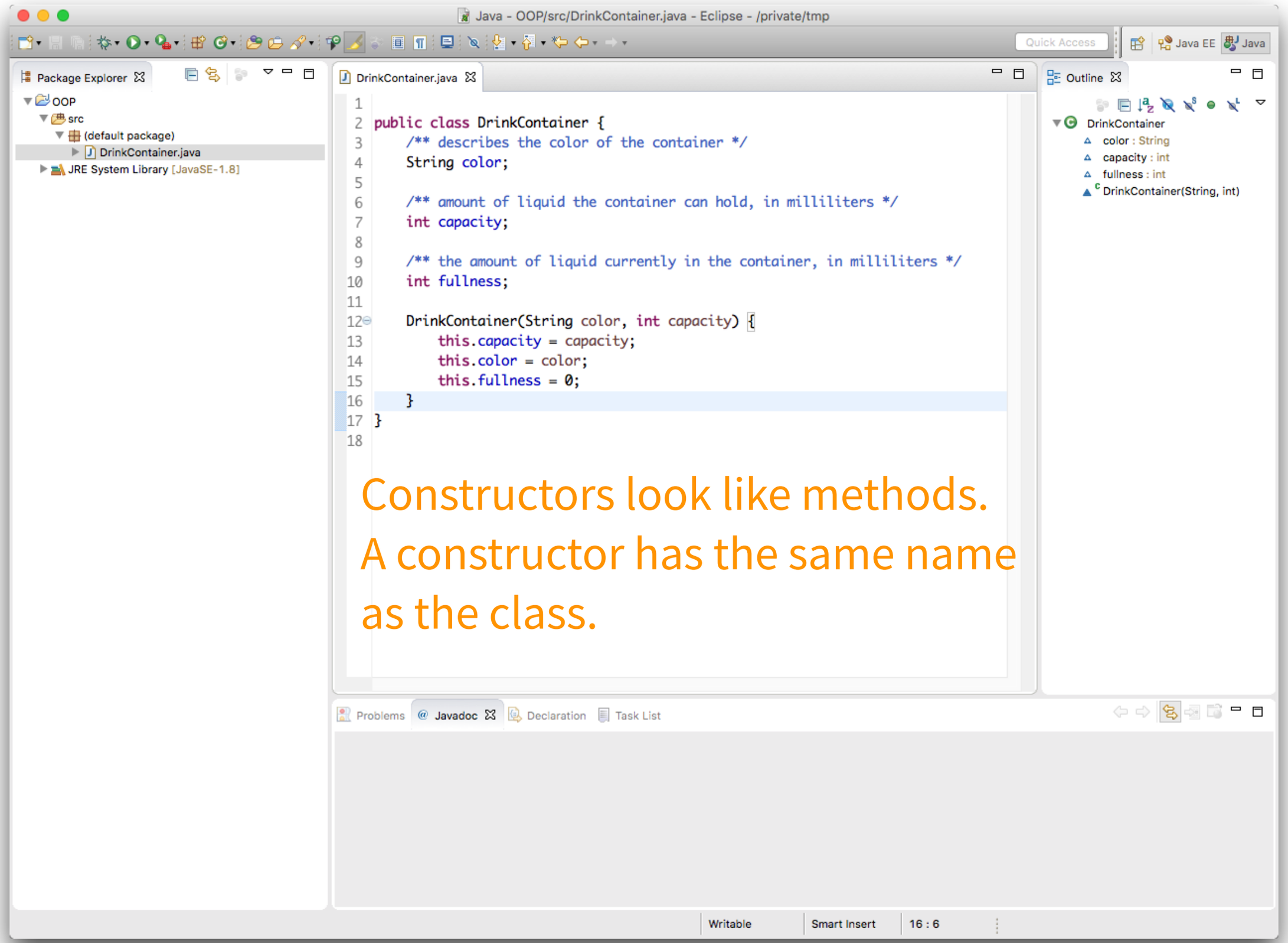


Good programming practice

Document your fields (using Javadoc).



A constructor initializes an object



Constructors look like methods.
A constructor has the same name as the class.

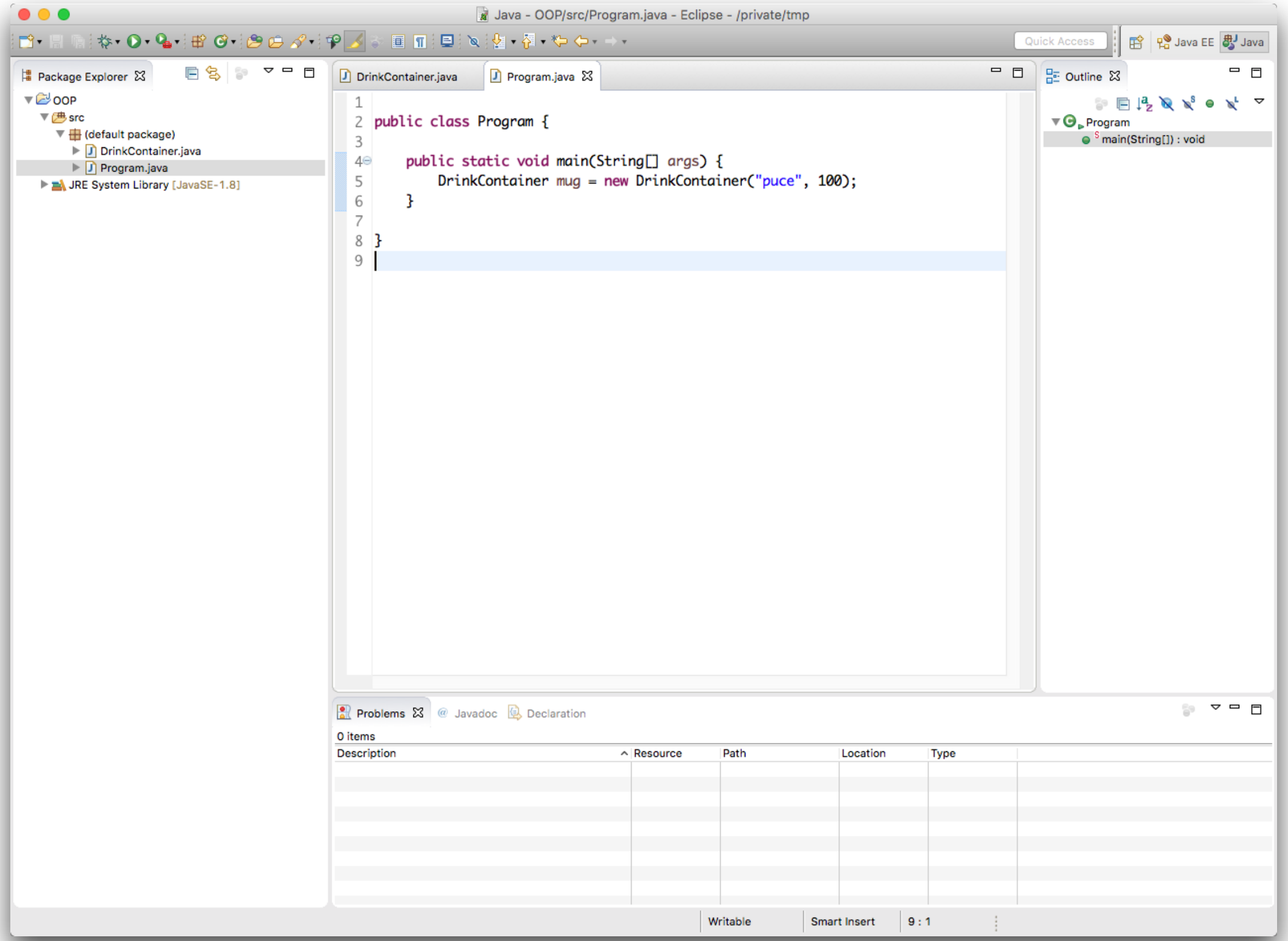
Good programming practice

Always use `this`.

It’s not a universally agreed-upon practice, but we’re going to follow it.

Use new to instantiate a Java object

Calls the constructor

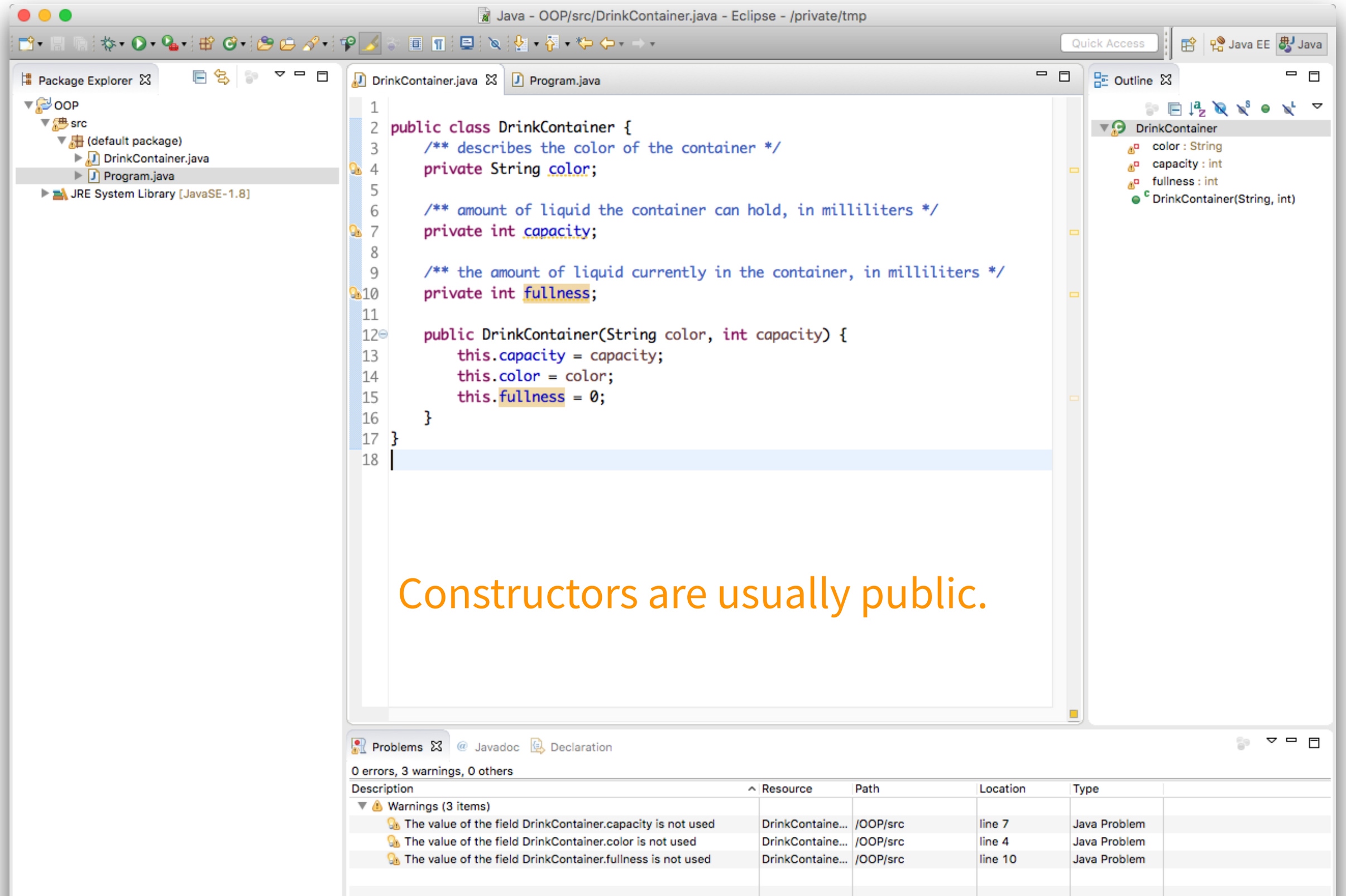


Good programming practice

Keep your `main` program separate from your class definitions.

Fields are usually private

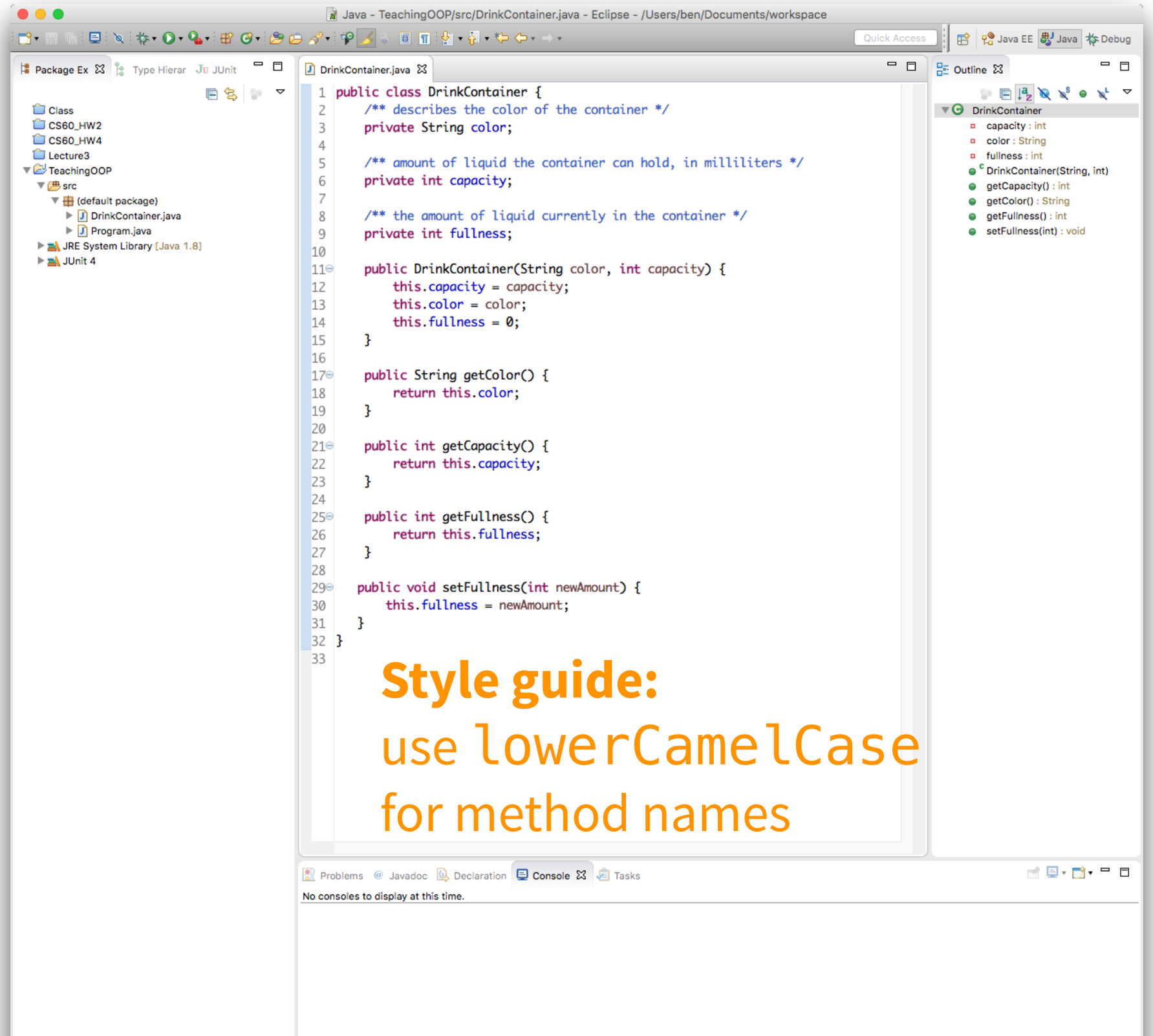
Fields are usually part of the **implementation** and should be hidden to the user.



Constructors are usually public.

Fields are accessed via `public` methods

We call these *accessor methods* (or *getters* & *setters*).



Style guide:
use `lowerCamelCase`
for method names

Good programming practice

Not every field needs accessors.

Good programming practice

Document your methods
(using Javadoc).

Good programming practice

Write tests first.

Good programming practice

Minimize the number of methods that access fields.

Instead, use existing methods (e.g., getters & setters).

It’s not a universally agreed-upon practice, but we’re going to follow it.

Java - TeachingOOP/src/DrinkContainer.java - Eclipse - /Users/ben/Documents/workspace

Quick Access | Java EE | Java | Debug

Package Ex | Type Hierar | JUnit

- Class
- CS60_HW2
- CS60_HW4
- Lecture3
- TeachingOOP
  - src
    - (default package)
      - DrinkContainer.java
      - DrinkContainerTests.java
      - Program.java
  - JRE System Library [Java 1.8]
  - JUnit 4

DrinkContainer.java | DrinkContainerTests.java

```
8     /** the amount of liquid currently in the container */
9     private int fullness;
10
11    public DrinkContainer(String color, int capacity) {
12        this.capacity = capacity;
13        this.color = color;
14        this.fullness = 0;
15    }
16
17    public String getColor() {
18        return this.color;
19    }
20
21    public int getCapacity() {
22        return this.capacity;
23    }
24
25    public int getFullness() {
26        return this.fullness;
27    }
28
29    /**
30     * Sets the new liquid amount for the mug. If the new amount exceeds the
31     * mug's capacity, the resulting fullness is the capacity. If the new amount
32     * is negative, the resulting fullness is unchanged.
33     *
34     * @param newAmount
35     */
36    public void setFullness(int newAmount) {
37        // If the new amount exceeds the mug's capacity, the resulting fullness
38        // is the capacity.
39        if (newAmount > this.getCapacity()) {
40            this.fullness = this.getCapacity();
41        }
42
43        else if (newAmount >= 0) {
44            this.fullness = newAmount;
45        }
46
47        // If the new amount is negative, the resulting fullness is unchanged.
48    }
49 }
50
```

Outline

- DrinkContainer
  - capacity : int
  - color : String
  - fullness : int
  - DrinkContainer(String, int)
  - getCapacity() : int
  - getColor() : String
  - getFullness() : int
  - setFullness(int) : void

Problems | Javadoc | Declaration | Console | Tasks

No consoles to display at this time.

Implement the `fill` method

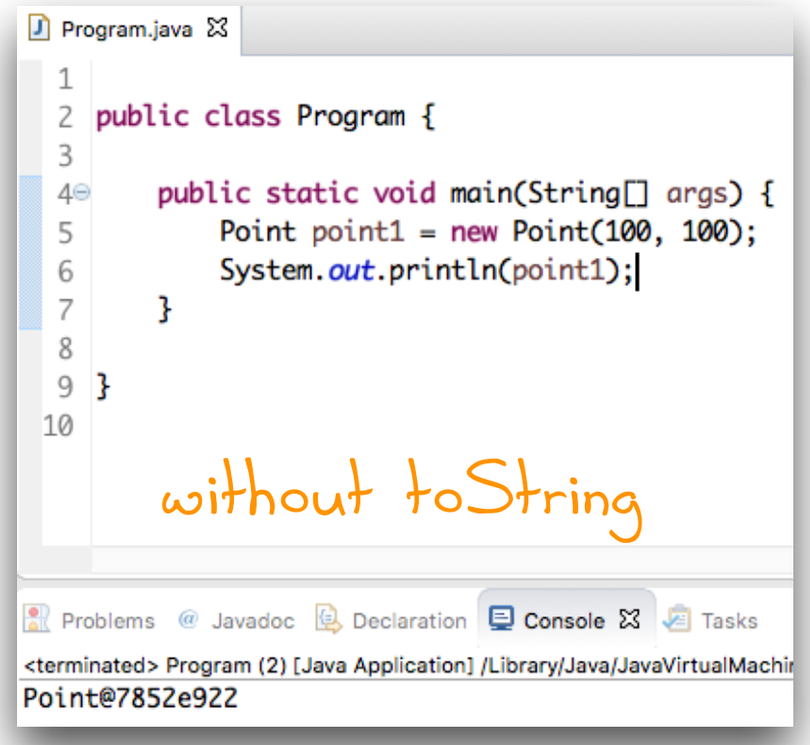
The `fill` method should fill the container to capacity.

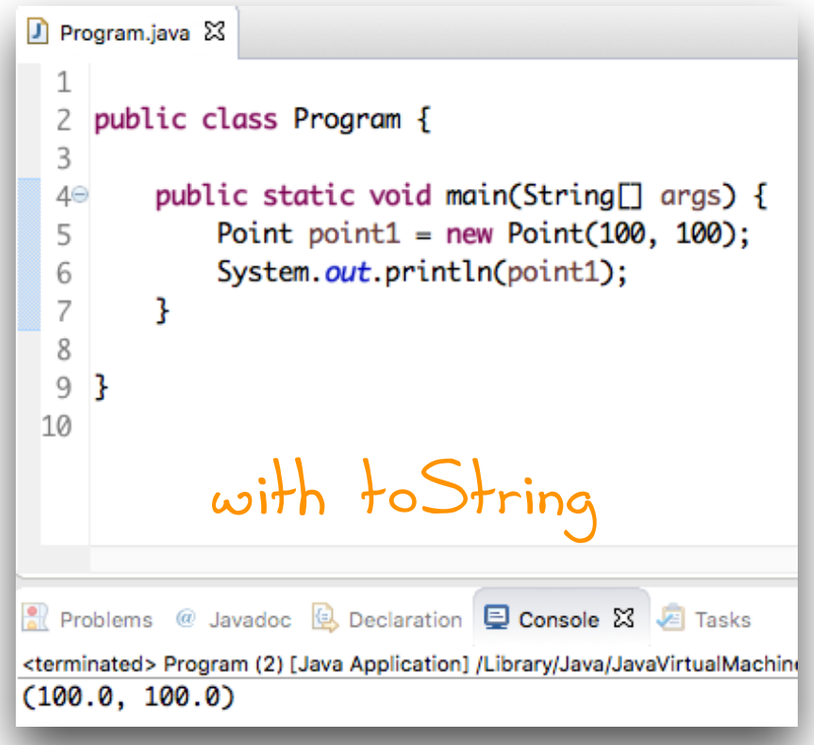
Write code for the entire method,
using all the good programming practices we've discussed.

Good programming practice

Write a `toString` method

The method takes no arguments and returns a `String`.






```
public String toString() {
    return "(" + this.getX() + ", " + this.getY() + ")";
}
```

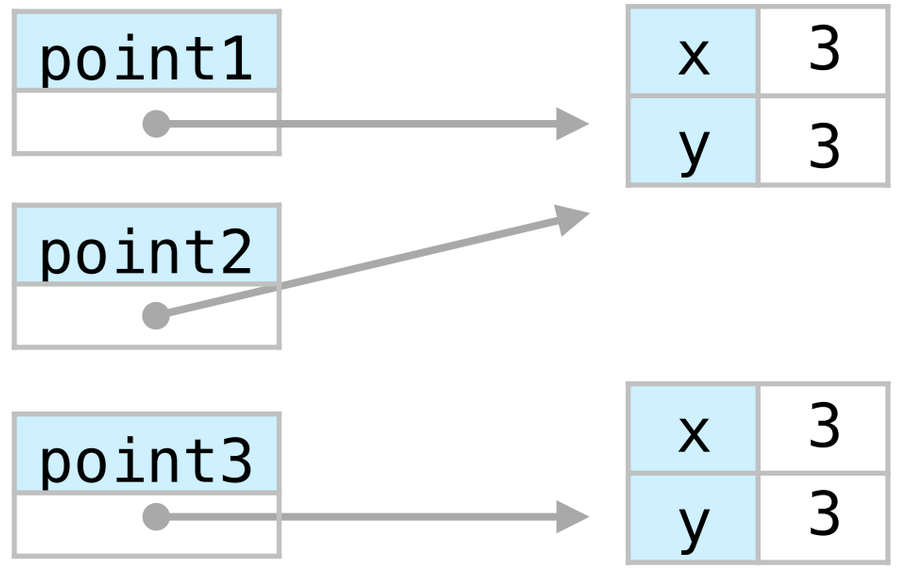
Objects and equality

Which of the following is true, and why?

```
Point point1 = new Point(3, 3);

Point point2 = point1;

Point point3 = new Point(3,3);
```




```
point1 == point2  ✔

point1 == point3

point2 == point3
```

Watch out!

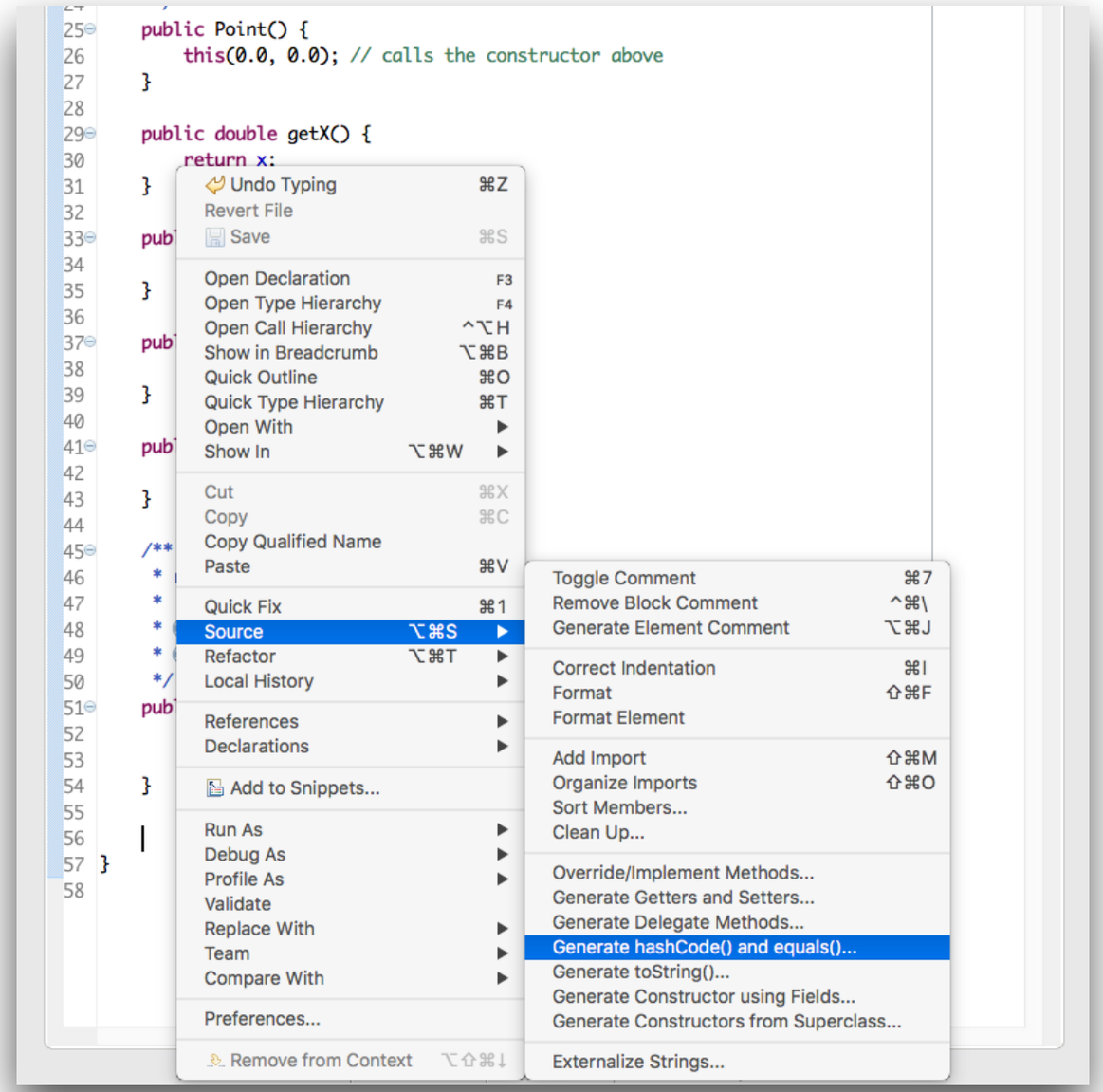
The implementer must provide `equals`

Otherwise, it may default to reference equality (which is probably not what we want).

Good programming practice

Auto-generate `equals` (and `hashcode`)

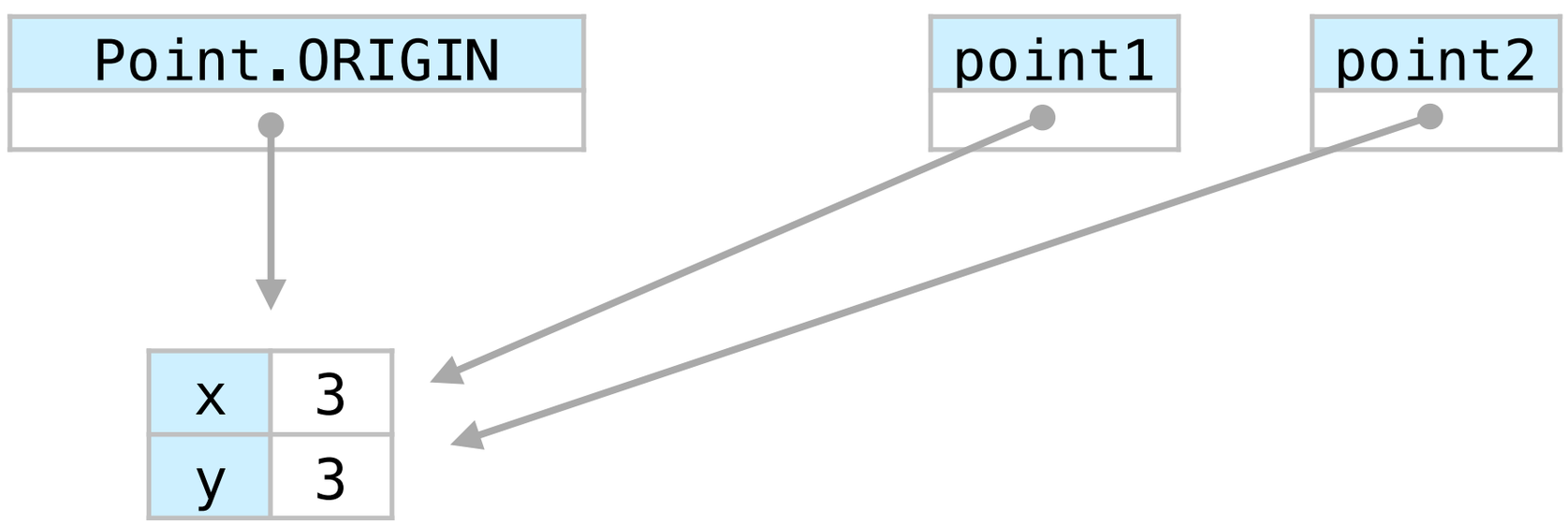
We normally like to write as much code ourselves as possible. But these methods are … special.



A class's static field values are the **same** for all instances of the class.

```
Point point1 = Point.ORIGIN;
Point point2 = Point.ORIGIN;
```

Style guide:
use ALL_UPPER_CASE
for static field names



Good programming practice

Always refer to a static method via the class.

Never refer to a static method via an object.

```
2  public class Program {
3
4      public static void main(String[] args) {
5          Point.myStaticMethod(10);          // like this
6
7          Point point1 = new Point(100, 100);
8          point1.myStaticMethod(10);         // not like this!
9      }
10
11 }
```

```
2  public class Program {
3
4      public static void main(String[] args) {
5          Point.myStaticMethod(10);          // like this
6
7          Point point1 = new Point(100, 100);
8  The static method myStaticMethod(double) from the type Point should be accessed in a static way
9      }
10
11 }
```

A class's static methods don't need an instance (and they can't use `this`).

```
58    public static double myStaticMethod(double value) {
59        return this.getX();    // makes no sense
60    }
```

```
58    public static double myStaticMethod(double value) {
59        return this.getX();    // makes no sense
          Cannot use this in a static context
60    }
```

Watch out!

Java initializes fields with a default value.

The default value of non-primitive fields is `null`.

The default value of a primitive field depends on its type.

Rectangle.java

```
1
2  public class Rectangle {
3      private Point topLeft;
4      private Point topRight;
5
6      public Point getTopLeft() {
9
10     public void setTopLeft(Point topLeft) {
13
14     public Point getTopRight() {
17
18     public void setTopRight(Point topRight) {
21
22     public String toString() {
25
27     public int hashCode() {
35
37     public boolean equals(Object obj) {
64 }
65
```

Program.java

```
1
2  public class Program {
3
4      public static void main(String[] args) {
5          Rectangle myRectangle = new Rectangle();
6          System.out.println(myRectangle.getTopLeft().getX());
7      }
8
9  }
10
```

Problems @ Javadoc Declaration Console Tasks

<terminated> Program (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Cor

```
Exception in thread "main" java.lang.NullPointerException
        at Program.main(Program.java:6)
```

Good programming practice

Provide good constructors.

Rectangle.java

```
1
2  public class Rectangle {
3      private Point topLeft;
4      private Point topRight;
5
6      public Rectangle(Point topLeft, Point topRight) {
7          this.topLeft = topLeft;
8          this.topRight = topRight;
9      }
10
11     /**
12      * No-argument constructor makes a rectangle at coordinates
13      *       (0, 0) and (0, 0)
14      */
15     public Rectangle() {
16         this(Point.ORIGIN, Point.ORIGIN);
17     }
18
```

how to call the two-argument constructor

Program.java

```
1
2  public class Program {
3
4      public static void main(String[] args) {
5          Rectangle myRectangle = new Rectangle();
6          System.out.println(myRectangle.getTopLeft().getX());
7      }
8
9  }
```

Problems @ Javadoc Declaration Console Tasks

<terminated> Program (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Cont

```
0.0
```

Object-oriented programming languages **differ** in:

- how the programmer specifies an object's **interface**
- how the programmer specifies an object's **implementation**
- how objects are **created**, **initialized**, **queried**, and **updated**
- **encapsulation** mechanism
  how strictly the language *enforces* the separation between interface & implementation

Encapsulation is a social construct

There is no public, protected, private in Python

If a field or method of a class is *not* part of the interface, prepend the name of that field / method with an underscore.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self._age = age

def setAge(self, newAge):
   if newAge < self._age:
      raise ValueError, "You can't get younger! (sorry)"

   self._age = newAge

def getAge(self):
  return self._age
```

Don't touch

NOT part of the interface

(Python does not try to enforce

Encapsulation is a social construct

Java has some language features that can help

If a field or method of a class is *not* part of the interface, use `private`.

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    ...
```

Don't touch
NOT part of the interface!
(Java enforces at compile time)