Java overview
(continued)
Given these declarations,

```java
Point point1 = new Point(3, 3);
Point point2 = point1;
Point point3 = new Point(3, 3);
```

Which of the following is true (and why)?

- `point1 == point2`
- `point1 == point3`
- `point2 == point3`

(Your response)
Use new to instantiate a Java object

Calls the constructor

```java
public class Program {
    public static void main(String[] args) {
        DrinkContainer mug = new DrinkContainer("puce", 100);
    }
}
```
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
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.
/**
 * the amount of liquid currently in the container */
 private int fullness;

 public DrinkContainer(String color, int capacity) {
     this.capacity = capacity;
     this.color = color;
     this.fullness = 0;
 }

 public String getColor() {
     return this.color;
 }

 public int getCapacity() {
     return this.capacity;
 }

 public int getFullness() {
     return this.fullness;
 }

 /**
 * Sets the new liquid amount for the mug. If the new amount exceeds the
 * mug's capacity, the resulting fullness is the capacity. If the new amount
 * is negative, the resulting fullness is unchanged.
 * @param newAmount
 */
 public void setFullness(int newAmount) {
     // If the new amount exceeds the mug's capacity, the resulting fullness
     // is the capacity.
     if (newAmount > this.getCapacity()) {
         this.fullness = this.getCapacity();
     }

     else if (newAmount >= 0) {
         this.fullness = newAmount;
     }

     // If the new amount is negative, the resulting fullness is unchanged.
 }
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`.

```java
public String toString() {
    return "(" + this.getX() + ", " + this.getY() + ")";
}
```
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.
A class’s static methods don’t need an instance (and they can’t use this).
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.
Good programming practice

Provide good constructors.

```java
public class Rectangle {
    private Point topLeft;
    private Point topRight;

    public Rectangle(Point topLeft, Point topRight) {
        this.topLeft = topLeft;
        this.topRight = topRight;
    }

    /**
     * No-argument constructor makes a rectangle at coordinates
     * (0, 0) and (0, 0)
     */
    public Rectangle() {
        this(Point.ORIGIN, Point.ORIGIN);
    }
}
```

how to call the two-argument constructor

```java
public class Program {
    public static void main(String[] args) {
        Rectangle myRectangle = new Rectangle();
        System.out.println(myRectangle.getTopLeft().getX());
    }
}
```
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.

```python
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.

```java
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)