OOP in Java
Prior experience: programming languages

- Assembly
- Racket
- Python
- Java

Experience levels: none, lots

- Soon
- Later
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...

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

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Objects**
- String • LinkedList
- other library & user-defined types...

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

Java directly stores primitive values.

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";
<table>
<thead>
<tr>
<th></th>
<th>primitives</th>
<th>objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable stores the value</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>variable stores a reference</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>supports (==)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>supports .equals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>we can define new kinds</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>type name starts with lower-case letter</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>type name starts with upper-case letter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Objects can hold variables, which store references to objects.
- Objects support the `==` operator, but it's probably not what you want.
- Objects support the `equals` method.
- We can define new classes and objects.
- Type names start with an upper-case letter for objects.
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

Objects are like...

cookies
A class is like...

factory

Objects are like...

cars
A class is like...

factory

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
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());
    }
}

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 \(\approx\) columns
- a class defines the names and types (but not the values) of fields

Objects \(\approx\) rows
- each object has specific values for its field

<table>
<thead>
<tr>
<th></th>
<th>color</th>
<th>capacity</th>
<th>fullness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colleen’s mug</td>
<td>blue</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Ben’s jug</td>
<td>puce</td>
<td>1000</td>
<td>500</td>
</tr>
<tr>
<td>Zach’s coffee cup</td>
<td>white &amp; green</td>
<td>100000</td>
<td>0</td>
</tr>
</tbody>
</table>
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

class name and filename must match
Field definitions go at top of class

Style guide:
use lowerCamelCase for field names
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
/** 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() + ")";
}
```
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.

```java
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`).

```java
class MyClass {
    public static double myStaticMethod(double value) {
        return this.getX(); // makes no sense
    }
}
```
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);
    }
}
```

```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)