ANNOUNCEMENTS

Today is the last week of class
Today is Donut Monday (grab a donut!)
Wednesday is Fun Final Review (kahoot? who can say...)

WARMUP

what is object-oriented programming?
are you an object-oriented programmer?
can you jam with the console cowboys in cyberspace?

TODAY

OOP, encapsulation & inheritance

do you know anything about hackers?

record LEC-02

🍩 🍩 P

indiana, let it go

object
(instance of a **class**)

anatomy of a class (1/2)

```
class ClassName {
    VariableOneType variableOne;
    ...

    FunctionOneReturnType functionOneName(...) { ... }
    ...
}
```

- a **class** is (a blueprint for) a lil chunk of data that you can make elsewhere
  - a class may have any number of **variables** (fields)
    - `int foo; // objects of this class have an int called foo`
  - a class may have any number of **functions** (methods)
    - `int bar() { ... } // objects of class have function bar`

anatomy of a class (2/2)

```
class Vector2 {
    // instance variables
    double x;
    double y;

    // constructor
    Vector2(double x, double y) { ... }

    // instance methods
    double length() { ... }
    ...
}
```

an **object** is **an instance of a class**

```
// v is an instance of the Vector2 class
// v is a Vector2 object
// "v is a Vector2"
Vector2 v = new Vector2();
```

object-oriented programming (OOP)
note: jim maybe has opinions (who's to say)

object-oriented programming (OOP)

- **object-oriented programming** means *thinking* in terms of nouns
  - "how can i break down this problem into classes/objects?"

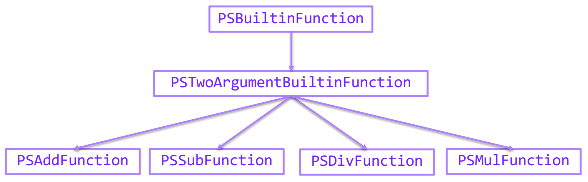
object-oriented programming (OOP)

- **object-oriented programming** is NOT just *having* classes/objects
  - recall, a **class** is just (a blueprint for) a lil chunk of data
  - rather, OOP means my problem-solving is *oriented* around objects
    - ...instead of, for example, data (**data-oriented design**)

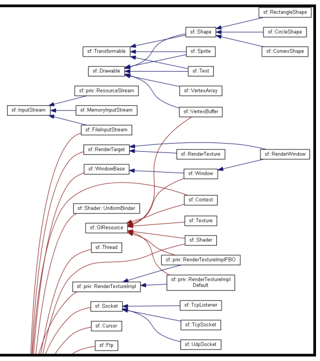
object-oriented programming

- **example:** to implement an Object-Oriented PostScript interpreter...
  - `class PSInterpreter`
  - `class PSProgram`
  - `class PSStack`
  - `class PSMap`
  - `class PSBuiltinFunction`
  - `class PSTwoArgumentBuiltinFunction extends PSBuiltinFunction`
  - `class PSAddFunction extends PSTwoArgumentBuiltinFunction`
  - `class PSSubFunction extends PSTwoArgumentBuiltinFunction`
  - `class PSMulFunction extends PSTwoArgumentBuiltinFunction`
  - `class PSDivFunction extends PSTwoArgumentBuiltinFunction`
  - `...`

unified modeling language (UML) diagram



real-world example:
SMFL



note that we still haven't written any actual code

we've made a *plan* for how to break the problem into objects

note: it can be hard to break problems into objects

Consider a very basic question: should a Message send itself?
'Sending' is a key thing I wish to do with Messages,
so surely Message objects should have a 'send' method, right?
If Messages don't send themselves, then some other object
will have todo the sending, like perhaps some not-yet-created Sender object.
Or wait, every sent Message needs a Recipient, so maybe instead Recipient
objects should have a 'receive' method.
This is the conundrum at the heart of object decomposition.
Every behavior can be re-contextualized by swapping around
the subject, verb, and objects.
Senders can send messages to Recipients;
Messages can send themselves to Recipients;
and Recipients can receive messages.
--Brian Will

so

it is very hard to break a problem into objects

but

it is also very *popular* to break a problem into objects

so let's learn some OOP 😳 👍

inheritance

one class can **inherit** from another

- a **child class** (**derived class**, **subclass**) inherits from its **parent class** (**base class**, **superclass**)
  - a child **inherits** (gets, has) its parents' variables and functions

```
// Inheritance: HW13 is an App
class HW13 extends App {
    // HW13 overrides loop()
    void loop() {
        if (keyPressed('a')) {
            ...
        }
    }
}
```

```
class App {
    Vector2 mousePosition;
    boolean keyPressed(...);
    void setup() { ... }
    void loop() { ... }
    ...
}
```

inheritance is convenient, but NOT fundamental

(except sometimes in Java)

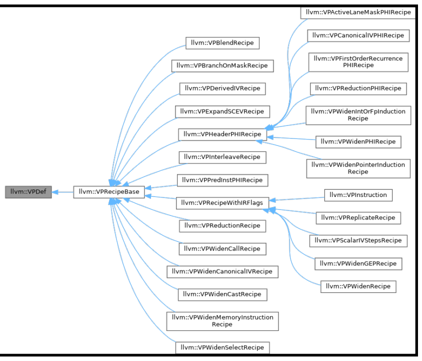
- instead of extending a class, we can store a reference to an instance of it
  - this is called **"composition"**
  - we will have to use the dot operator (a lot) more, but c'est la vie

```
// Inheritance: HW13 is an App
class HW13 extends App {
    // HW13 overrides loop()
    void loop() {
        if (keyPressed('a')) {
            ...
        }
    }
}
```

```
// Composition: HW13 has an App
class HW13 {
    App app;
    void loop() {
        if (app.keyPressed('a')) {
            ...
        }
    }
}
```

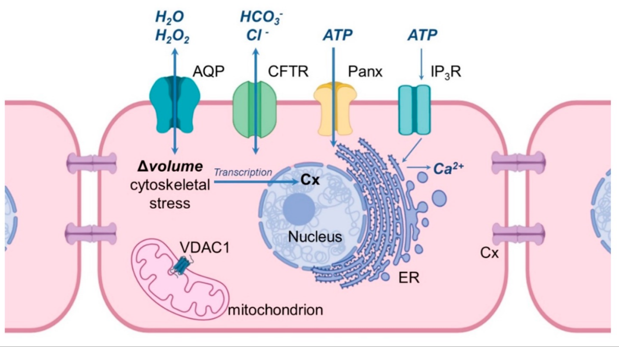
the fundamental point is that maybe it's nice to reuse code

maybe


llvm::VPDef
llvm::VPRecipeBase
llvm::VPBlendRecipe
llvm::VPBranchOnMaskRecipe
llvm::VPDerivedIVRecipe
llvm::VPExpandSCEVRecipe
llvm::VPHeaderPHIRecipe
llvm::VPInterleaveRecipe
llvm::VPPredInstPHIRecipe
llvm::VPRecipeWithIRFlags
llvm::VPReductionRecipe
llvm::VPWidenCallRecipe
llvm::VPWidenCanonicalIVRecipe
llvm::VPWidenCastRecipe
llvm::VPWidenMemoryInstructionRecipe
llvm::VPWidenSelectRecipe
llvm::VPActiveLaneMaskPHIRecipe
llvm::VPCanonicalIVPHIRecipe
llvm::VPFirstOrderRecurrencePHIRecipe
llvm::VPReductionPHIRecipe
llvm::VPWidenIntOrFpInductionRecipe
llvm::VPWidenPHIRecipe
llvm::VPWidenPointerInductionRecipe
llvm::VPInstruction
llvm::VPReplicateRecipe
llvm::VPScalarIVStepsRecipe
llvm::VPWidenGEPRecipe
llvm::VPWidenRecipe

final note: inheritance doesn’t simplify a problem
so much as i t s p r e a d s i t o u t
(and spreads it *around*?)

encapsulation



encapsulation

- **encapsulation** is the idea that a class should be like a "capsule"
  - the variables inside the capsule should be **`private`**
    - users of the class CANNOT touch them
  - for its users, the class should expose safe, **`public`** functions

```
void put(KeyType key, ValueType value) { ... }
ValueType get(KeyType key) { ... }
int size() { return this._size; }
```

```
ArrayList<KeyValuePair>[] buckets;
int _size;
```

encapsulated hashmap

note: this probably makes sense

the typical user of a hashmap shouldn't be messing with the **`private`** array

(and perhaps the exceptional user should write their own hashmap)

big idea: encapsulation can hide away messy, dangerous details

note: this is just a metaphor;
do NOT play with fire

however

encapsulation can maybe be taken too far

```
int _bullet;
```

```
bullet.age++;
```

```
bullet.setAge(bullet.getAge() + 1);
```

```
bullet.ageUp(); // ?
```

OOP considered maybe mildly frictious to prototyping

final note: encapsulation doesn't *add* functionality
encapsulation *removes* functionality

iterators

cool gradient in cow

course review
(big O & array)