Week13

-OOP

WARMUP

what is object-oriented programming?

are *you* an object-oriented programmer?

do you know *anything* about hackers?

can you jam with the console cowboys in cyberspace?!

ANNOUNCEMENTS

doughnuts on wednesday

PBS's Ghostwriter. Season 2 Episode 5

*NOTE: doughnuts on wednesday

indiana, let it go

[record lecture]

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 Point {
    // instance variables
    double x;
    double y;

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

    // instance methods
    double distanceTo(Point otherPoint) { ... }
    ...
}
```

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

```
// p is a reference to an instance of the Point class
// p is an instance of the Point class
// p is a Point object
// "p is a Point"
Point p = new Point();
```

object-oriented programming (OOP)

note: jim maybe has opinions but who's to sayyy

object-oriented programming (OOP)

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

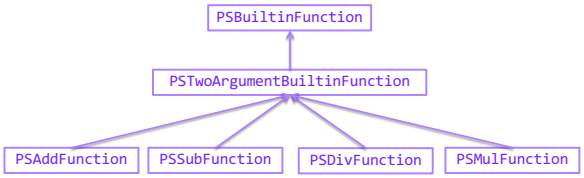
object-oriented programming (OOP)

- **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**)
    - ...functions (**functional programming**)

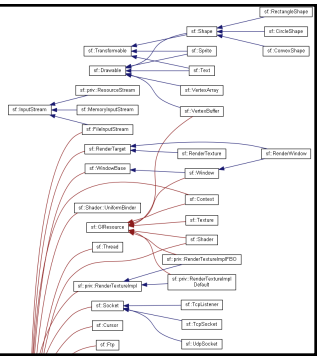
object-oriented programming

- **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

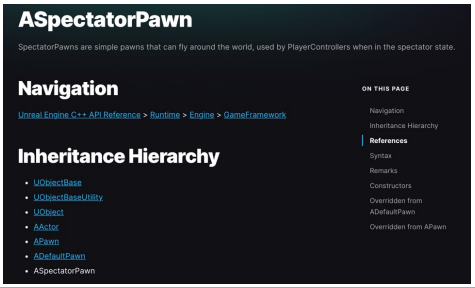
it is very hard to break a problem into objects

but

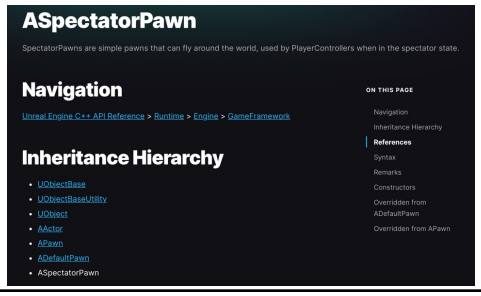
it is also very *popular* to break a problem into objects

so let's learn some OOP 🙂 👍

example: Unreal Engine API



example: PyTorch



ASpectatorPawn

SpectatorPawns are simple pawns that can fly around the world, used by PlayerControllers when in the spectator state.

Navigation

Unreal Engine C++ API Reference > Runtime > Engine > GameFramework

Inheritance Hierarchy

- UObjectBase
- UObjectBaseUtility
- UObject
- AActor
- APawn
- ADefaultPawn
- ASpectatorPawn

ON THIS PAGE

Navigation
Inheritance Hierarchy
References
Syntax
Remarks
Constructors
Overridden from ADefaultPawn
Overridden from APawn

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 a Cow (app)
class HW13 extends Cow {
    public static void main(...) {
        while (beginFrame()) {
            ...
        }
    }
}
```

```
class Cow {
    static boolean beginFrame(...);

    static double mouseX;
    static double mouseY;
    static boolean keyPressed(...);
    ...
}
```

inheritance is convenient, but NOT fundamental

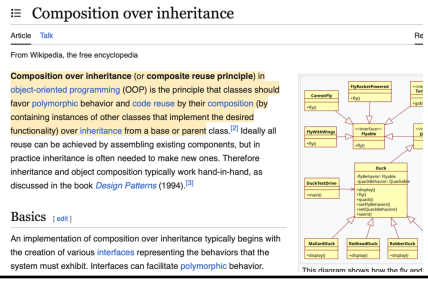
(except sometimes in Java)

- 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; they're fundamentally the same thing*

```
// Inheritance: HW13 is a Cow
class HW13 extends Cow {
    public static void main(...) {
        while (beginFrame()) {
            ...
        }
    }
}
```

```
// Composition: HW13 has a Cow
class HW13 {
    static Cow cow;
    public static void main(...) {
        while (cow.beginFrame()) {
            ...
        }
    }
}
```

***sidenote:** some people don't think agree with me that composition and inheritance are the same thing



Composition over inheritance

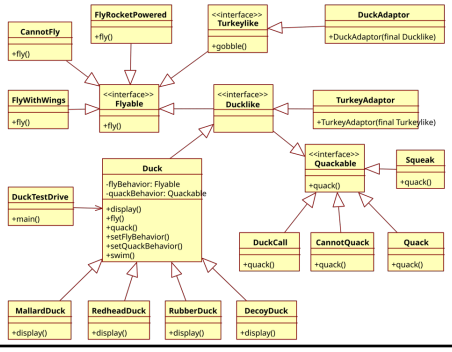
Article Talk

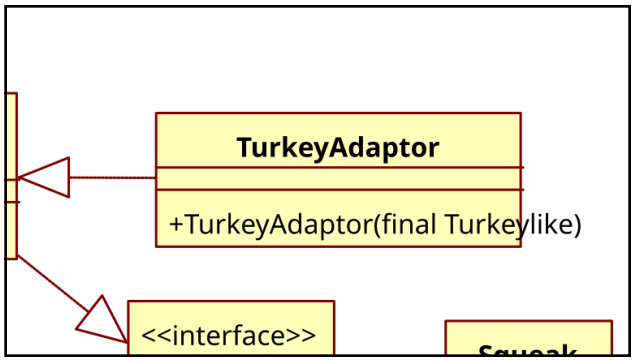
From Wikipedia, the free encyclopedia

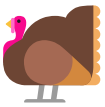
Composition over inheritance (or **composite reuse principle**) in object-oriented programming (OOP) is the principle that classes should favor polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) over inheritance from a base or parent class.[2] Ideally all reuse can be achieved by assembling existing components, but in practice inheritance is often needed to make new ones. Therefore inheritance and object composition typically work hand-in-hand, as discussed in the book *Design Patterns* (1994).[3]

Basics [edit]

An implementation of composition over inheritance typically begins with the creation of various interfaces representing the behaviors that the system must exhibit. Interfaces can facilitate polymorphic behavior.





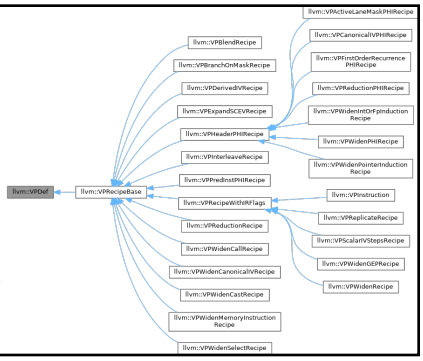


gobble gobble

anyway.

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

maybe



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*?)

down the rabbit hole...

```
public class ArrayList<E>
```

down the rabbit hole...

```
public class ArrayList<E> extends AbstractList<E>
                          implements List<E>,
                                     RandomAccess,
                                     Cloneable,
                                     java.io.Serializable
```

down the rabbit hole...

```
public abstract class AbstractList<E> extends AbstractCollection<E>
                                      implements List<E> {
```

down the rabbit hole...

```
public abstract class AbstractCollection<E> implements Collection<E> {
```

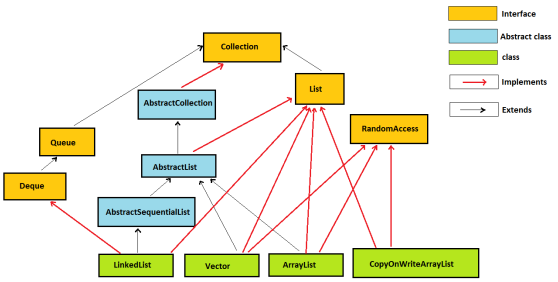
down the rabbit hole...

```
public interface Collection<E> extends Iterable<E> {
```

down the rabbit hole...

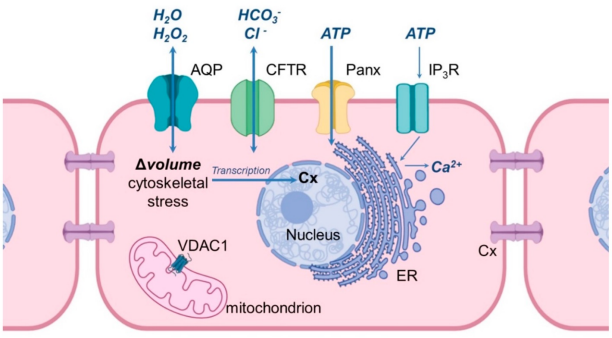
```
public interface Iterable<T> {


    /**
     * Returns an iterator over a set of elements of type T.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
}
```

down the rabbit hole...



gobble gobble

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

```
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

special topic:
conway's law

[O]rganizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.
—*Melvin E. Conway, How Do Committees Invent?*