# Announcements

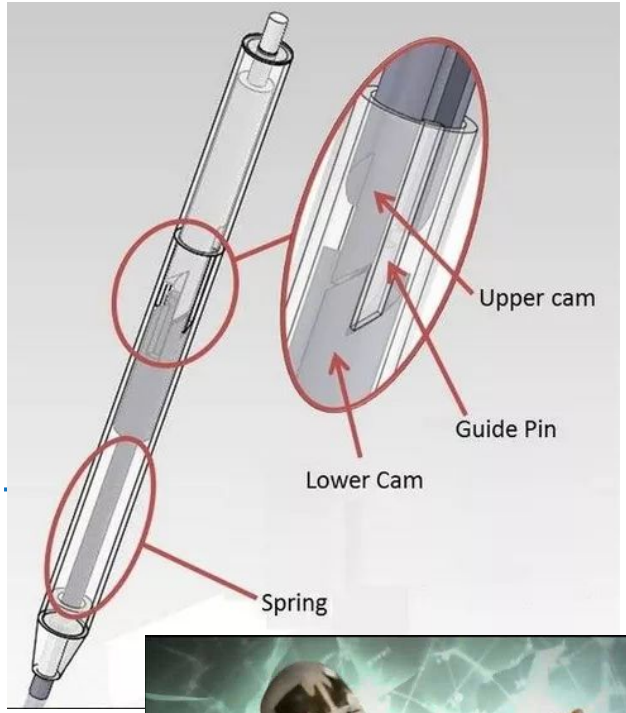
Start project 2 coding ASAP!

- If you haven't make sure you've watched these videos on getting started
- Phase 1 almost over (early deadline is **today**)
- Read the clarifications here: https://piazza.com/class/j9j0udrxjjp758?cid=1616
- Update your IntelliJ plugin to v1.0.21!

# CS61B, Spring 2018

Lecture 16: Programming Efficiently

- Programming Efficiently, APIs
- ADT Implementations
- Views




# 61B: Writing Efficient Programs

"An engineer will do for a dime what any fool will do for a dollar"

-- Paul Hilfinger

Efficiency comes in two flavors:

- Programming cost.
  - How long does it take to develop your programs?
  - How easy is it to read, modify, and maintain your code?
    - More important than you might think!
    - Majority of cost is in maintenance, not development!
- Execution cost (starting next week).
  - How much time does your program take to execute?
  - How much memory does your program require?

# Keeping Programming Costs Low

Some Java features discussed in 61B:

- Packages.
  - Good: Way of organizing a large project.
  - Bad: Import * is dangerous!
- Static type checking.
  - Good: Promotes code clarity, very explicit, can prevent some errors.
  - Bad: Very verbose
- Inheritance.
  - Good: Can reuse code.
  - Bad: Can become confusing!

# Keeping Programming Costs Low (Java Features)

Some Java features discussed in 61B:

- Packages.
  - Good: Keep code organized (in folders). Canonical names for classes and other things.
  - Bad: More work to compile and run.
- Static type checking.
  - Good: Speeds up runtime (no need to runtime type check). Catch errors early (before anybody runs the code).
  - Bad: More verbose code.
- Inheritance (implementation and interface inheritance).
  - Good: Interface inheritance allows subtype polymorphism. Implementation inheritance allows code reuse.
  - Bad: Have to implement all features of an interface (can be many). Makes code harder to read/understand. More that we'll discuss.

# Keeping Programming Costs Low (Programming Practices)

Modularity: How can I break my problem down into easily understood subproblems?

Invariants: What properties must my algorithm or data structure maintain?

- Many programmers only consider invariants implicitly. Some write them out.

Testing: Create automated code verification tools to bolster (but not prove) correctness.

- A general framework for proving that code is correct is impossible. It's not even possible to prove that a piece of code avoids going into an infinite loop (see the halting problem if you're curious).

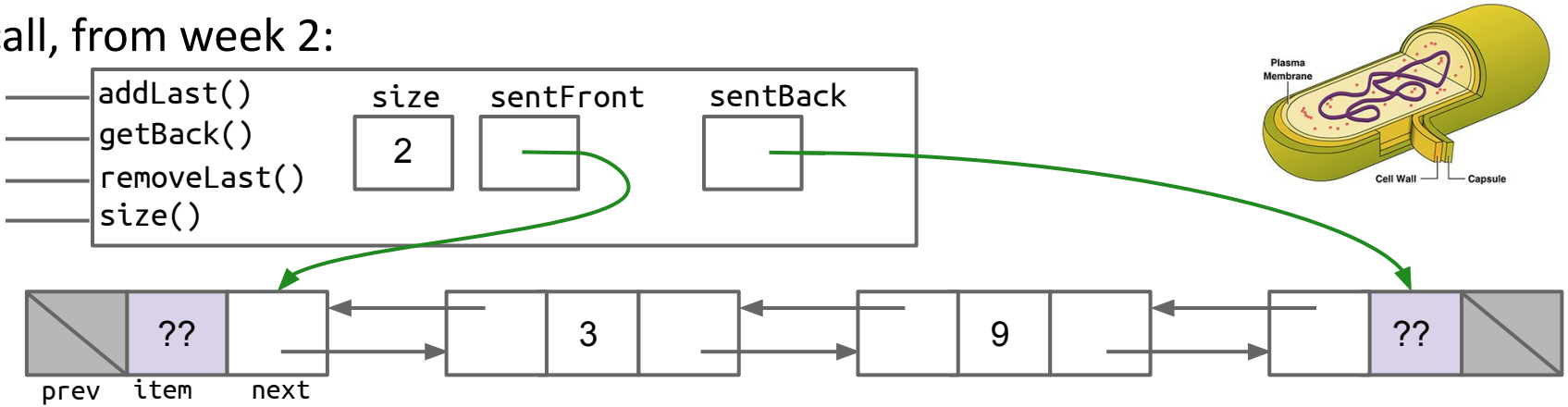
# Modules and Encapsulation (this slide, a blast from the past)

***Module***: A set of methods that work together as a whole to perform some task or set of related tasks.

A module is said to be ***encapsulated*** if its implementation is completely hidden, and it can be accessed only through a documented interface.

- Example: An Abstract Data Type describes an encapsulated data structure.

Recall, from week 2:



# ADTs and APIs

The API (Application Programming Interface) of an ADT is the list of constructors and methods, including an informal description of the effect of each.

- Not a precisely defined term. Some real world definitions of API are broader than ours.

API consists of ***syntactic*** and ***semantic*** specification.

- Compiler verifies that syntax is met.
- Tests help verify that semantics are correct.
  - Semantic specification usually written out in English (possibly including usage examples). Mathematically precise formal specifications are somewhat possible but not widespread.

# Example of a Semantic Specification

A semantic specification of project 2 from two years ago

- Link: https://docs.google.com/document/d/17PUw2EffgyU5_zQHZ8GV52EhS3NPwUyghFfX6EmJiS4/edit

This idea of hiding details from yourself or other programmers is **critically important**!

- This skill is a major divide between struggling and thriving 61B students.
- Struggling students try to fit everything in their head at once.

# Designing an API

One of the hardest parts about writing a program is deciding on the API.

- Project 2 will give you your only chance to do this in 61B.
- You'll find that it is a challenging task!

In the real world, API decisions are truly momentous.

- If you add a public method or variable, you are saying that it will be available… forever!
- Google and Oracle (owners of Java) lawsuit:
  - Google created their own implementation of some of Java's API.
  - Oracle claimed this was copyright infringement!
  - Google responded that their implementation was "fair use".
  - Lawsuit started in 2010, recently finished (Jury sided with Google)

# Designing an API

API design and data structure selection can be an iterative process.

- Nice to get it right the first time.
- But if you don't, it's often worth the trouble to refactor your code.
- The majority of your effort will be spent on the design!

Word of warning: `Map<String, Map<Integer, Double>> someMap;`

- Complex data type specifications like this can be a sign that you need to hide more from the class using the `someMap` variable.
- Example: `Map<String, TimeSeries<Double>> someMap;`

{FB: **2012**: 26, **2013**: 55, **2014**: 78, **2015**: 104,
AMZN: **1997**: 4.5, **1998**: 53, **1999**: 106, **2000**: 15, **2001**: 10,
NFLX: **2003**: 0.77, **2004**: 1.65, **2005**: 3.96, **2006**: 3.74,
GOOGL: **2004**: 96 **2005**: 233, **2006**: 230, **2007**: 300, **2008**: 295}

{FB: TimeSeries@f83f9182
AMZN: TimeSeries@087d7f33
NFLX: TimeSeries@7caaad29,
GOOGL: TimeSeries@99ffe71c}

# API Design Issues. See CS 169 for more.

Potential pitfalls.

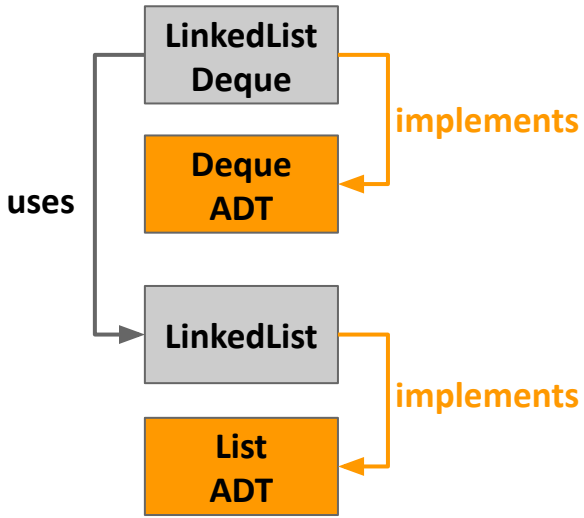
- Too hard (or impossible) to implement.
- Too hard to use, e.g. arguably JavaFX.
- Too general, e.g. the God object. Does too much on its own.
- Too specific, e.g. many libraries for making graphical plots (matplotlib).
- Too narrow, e.g. our early SList had no get(i) method.
- Too wide, e.g. the get(i) method in Deque.
- Too wide, e.g. the java.util.Stack class (more soon).

Helpful motto: “Provide to clients the methods they need and no other.” - Joe Armstrong

# ADT Implementations

# ADT Implementation

- Recall that ***ADTs*** are implemented using ***data structures*** (think List vs. Array).
  - Often implemented using existing data structures.
- Examples:
  - LinkedListDeque from project 1 **>>>**
  - In project 2, you are implementing a **Grid** ADT with existing data structures.
- **Use vs. Implementation**



# Special Case ADTs

Sometimes, ADTs are special cases of other ADTs. Three common special cases for Lists:

- Queue (also known as a line in America): Supports **enqueue** and **dequeue**.
  - **enqueue**: puts an item at end of queue
  - **dequeue**: removes and returns item at front of queue.
  - Also called a FIFO (first-in, first-out) List.
- Stack: Support **push** and **pop**.
  - **push**: puts things on 'top' of stack
  - **pop**: takes things off the top.
  - Also called a LIFO (last-in, first-out) List.
- Deque: Support **addFront**, **addBack**, **getFront**, **getBack**.

# Implementing Special Cases: Extension, Delegation, Adaptation

Suppose we have a List of some type, say a LinkedList. How do we get a Stack?

# Implementing Special Cases: Extension, Delegation, Adaptation

Two natural approaches:

- **Extension**:

```
public class ExtensionStack<Item> extends LinkedList<Item> {
    public void push(Item x) {
        add(x);
    }
}
```

Careful about has-a vs. is-a relationship!
More soon.

- **Delegation**:

```
public class DelegationStack<Item> {
    private LinkedList<Item> L = new LinkedList<Item>();
    public void push(Item x) {
        L.add(x);
    }
}
```

# Implementing Special Cases: Extension, Delegation, Adaptation

Another (more exotic) approach:

- **Adaptation**:

```
public class StackAdapter<Item> {
    private List L;
    public StackAdapter(List<Item> worker) {
        L = worker;
    }

    public void push(Item x) {
        L.add(x);
    }
}
```

# Designing ADTs

# Is-A vs. Has-A

Is-A:

- Examples:
  - A `Square` is-a `Rectangle`.
  - An `ArrayList` is-a `List`.
- In linguistics, known as *hyponymy*. `ArrayList` is a ***hyponym*** of `List`.

Has-A:

- Examples:
  - `Animals` have-a `Leg`.
  - `GuitarString` has-a `ArrayRingBuffer`.
- In linguistics, known as *holonymy*. `Animals` is a ***holonym*** of `Leg`.

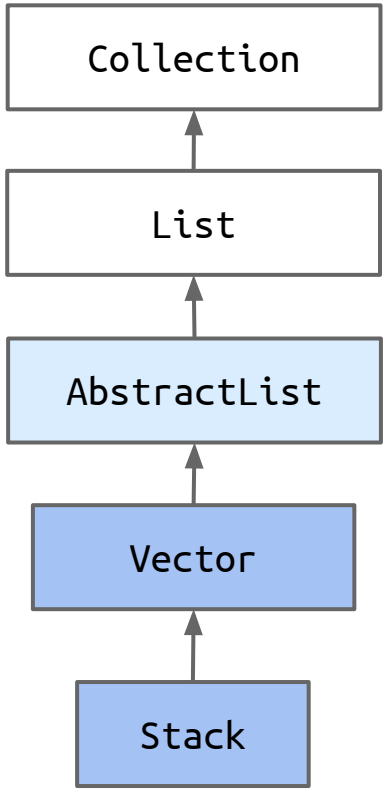
**Inheritance** relationships should ALWAYS be **is-a** relationships. Can be subtle.

# The `java.util.Stack`: A Prime Example in Bad Design

Created using extension instead of delegation.

- Authors asserted that a Stack IS-A Vector.

https://www.amazon.com/dp/B001DZTJRQ




Resulting API:

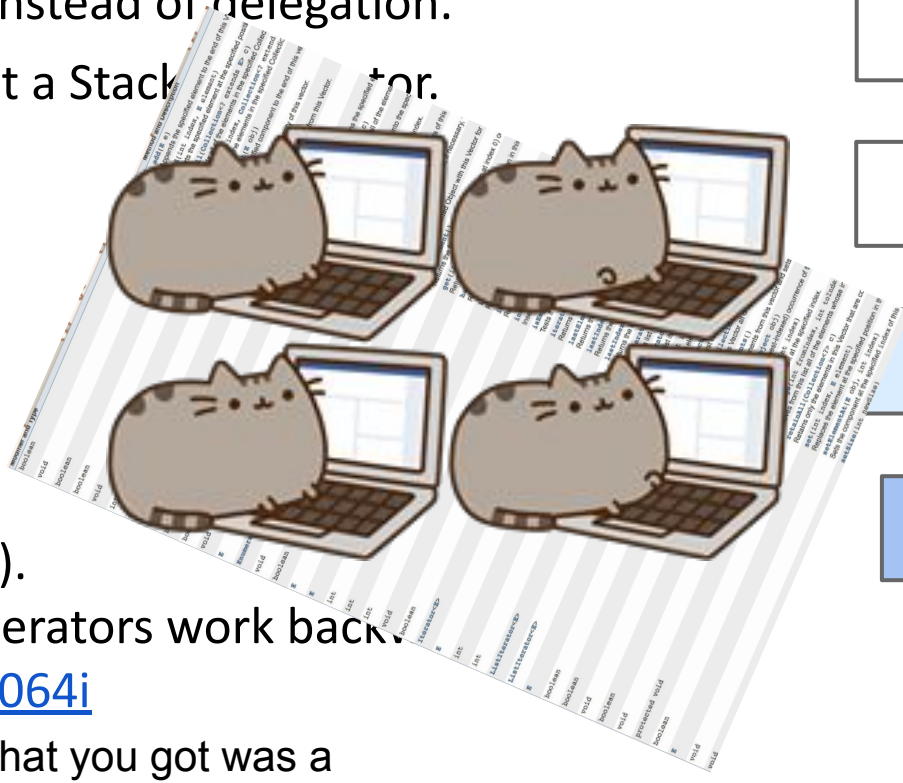
- Is complex (too wide).
- Is counterintuitive (iterators work backwards).
  - http://goo.gl/rU064i

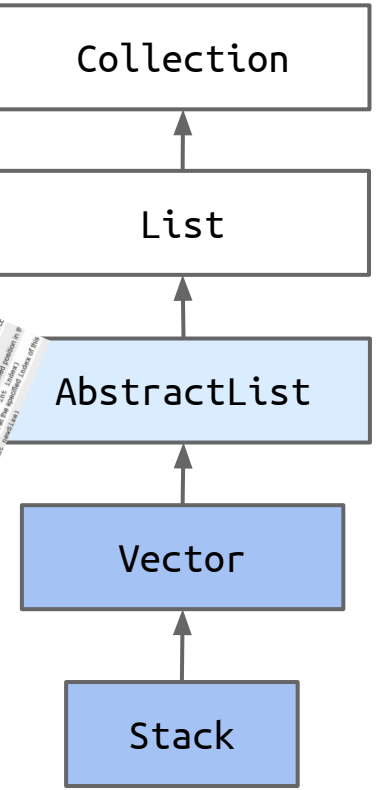
"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."
- Joe Armstrong (on OOP)

# The `java.util.Stack`: A Prime Example in Bad Design

Created using extension instead of delegation.

- Authors asserted that a Stack is a Vector.






Resulting API:

- Is complex (too wide).
- Is counterintuitive (iterators work backwards).
  - http://goo.gl/rU064i

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."
- Joe Armstrong (on OOP)

# An Even More Subtle Inheritance Pitfall

Classic example: A Square is-a Rectangle.

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w; height = h;
    }

    /** Sets the width. */
    public void setWidth(int w) {
        width = w;
    }

    /** Sets the height. */
    public void setHeight(int h) {
        height = h;
    }
```

```
public class Square extends Rectangle { ...
```

Deep arguments about program and programming language design result. See the circle-ellipse problem for more.

# Implementation Inheritance Breaks Encapsulation

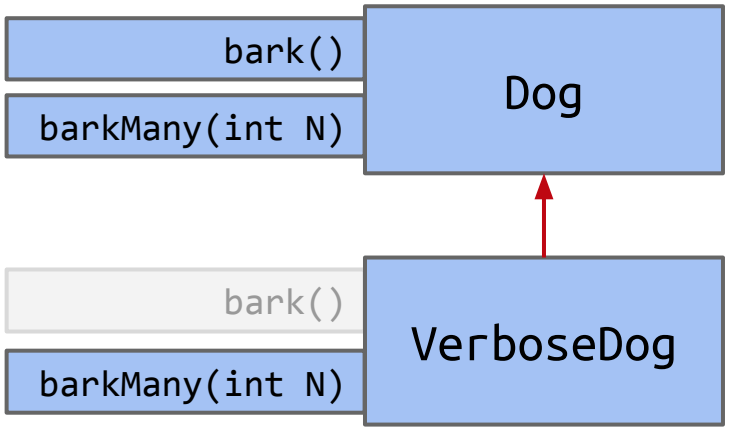
What would vd.barkMany(3) output?

**c. Something else.**

- Gets caught in an infinite loop!

(assuming vd is a Verbose Dog)

```
public void bark() {
    barkMany(1);
}
public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}
```




```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark(); ← calls inherited bark method
    }
}
```

# Delegation vs. Extension

As Josh Bloch puts it in **Effective Java**:

- It is safe to use inheritance within a package, where the subclass and the superclass are under the control of the same programmers.
- It is safe to extend classes specifically designed and documented for extension.
- Inheriting from ordinary concrete classes across package boundaries is dangerous.

When in doubt: Use delegation and write simple forwarding methods for each method.

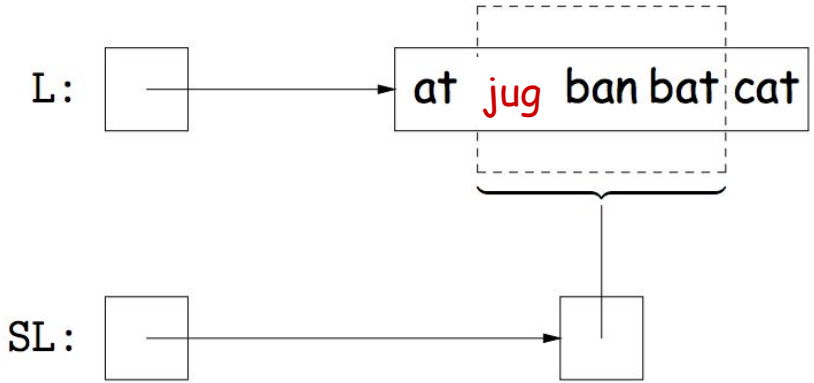
# Views

# Views into Lists

An alternative representation of an existing object.

- Concrete objects
- Access is limited
- But changes mutate the underlying object!

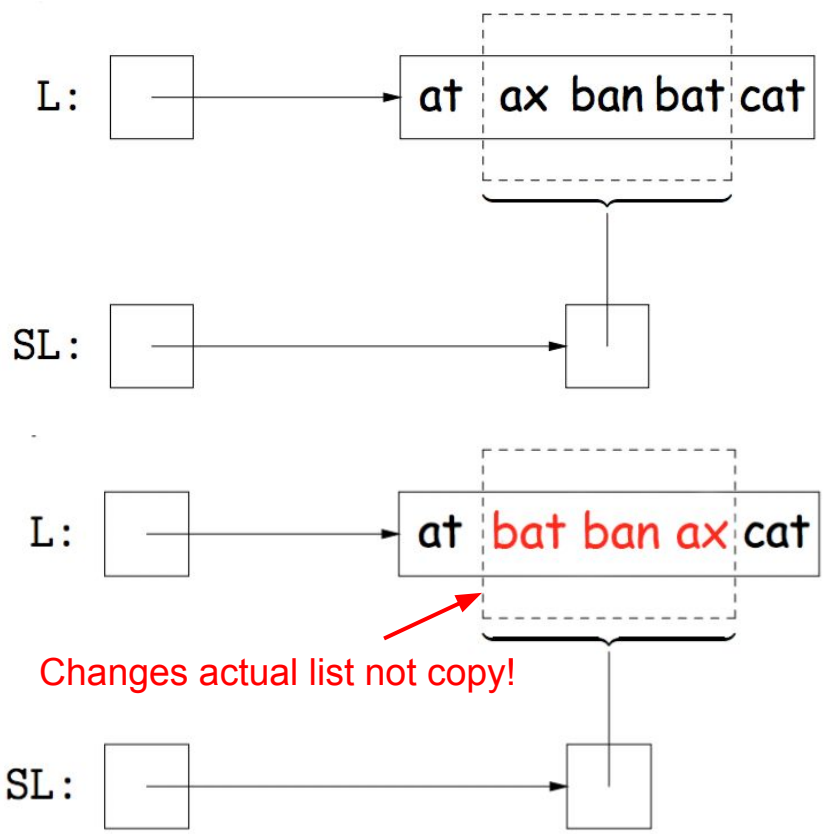
```
/** Create an ArrayList. */
List<String> L = new ArrayList<>();
/** Add some items. */
L.add("at"); L.add("ax"); ...
/** subList me up fam. */
List<String> SL = l.subList(1, 4);
/** Mutate it. */
SL.set(0, "jug");
```



# Views into Lists

Allows for general functions

```
/** Reverses a List */
void reverse(List<String> l) {...}
/** Reverse the sublist no cases */
reverse(SL);
```



# Views into Lists (Implementation)

How do you return an actual List but still have it affect another List? **Access methods!**

Code is taken from AbstractList w/ exceptions removed

Returns an actual List type

Only store the range of the view

The get/add method uses the surrounding class's get/add methods but limits access to the constructed range.

```
List<Item> sublist(int start, int end) {
  return new this.Sublist(start, end);
}

private class Sublist extends AbstractList<Item> {
  private int start, end;
  Sublist(int start, int end) {   ...   }

  public int size() { return end-start; }

  public Item get(int k) //exception if start+k >= end
    { return AbstractList.this.get(start+k); }

  public void add(int k, Item x) {
    { AbstractList.this.add(start+k, x); end += 1; }
  ...
}
```

# Views into Maps

The Map interface provides a number of 'Views' into the Map.

- Views can also save time and space.

```
public interface Map<Key, Value> {
    ...
    /* VIEWS */
    /** The set of all keys. */
    Set<Key> keySet();
    /** The multiset of all values. */
    Collection<Value> values();
    /** The set of all (key, value) pairs. */
    Set<Map.Entry<Key,Value>> entrySet();
    ...
}
```

Nested interface!

# Implementation Specific Views

Occasionally, implementation details may allow for views that are too difficult to implement for an abstract type.

Example:

- The Map has a **`keySet()`** method. Iteration on the **`keySet()`** may show keys in any order.
- The TreeMap class has a **`navigableKeySet()`** method. Iteration on the **`navigableKeySet()`** gives keys in order.
  - Inefficient with a HashMap, so HashMap does not have this method.

# The Punchline

APIs are hard to design.

- Proper design pays massive dividends.
- Strive always for code clarity.
- Inheritance is a powerful but dangerous tool.

For the next few weeks: Common high performance implementation some of the most common ADTs. Analysis of those implementations.

- Lists
- Sets
- Maps
- PriorityQueues
- DisjointSets

# Citations

Title images:

- Ballpoint pen diagram: https://qph.is.quoracdn.net/main-qimg-ab1f27c758b4c3074800cc0284c93230?convert_to_webp=true
- Dude from ICP: http://goo.gl/gZZwq9