Practical Polymorphism, and Intro Iterators

Quote of the week:
“Everybody has a secret world inside of them. All of the people of the world, I mean everybody. No matter how dull and boring they are on the outside, inside them they’e all got unimaginable, magnificent, wonderful, stupid, amazing worlds. Not just one world. Hundreds of them. Thousands maybe.”

If you haven't started project 1…

- Uh…
- Uh oh.
- You should start today!

Labs are now mostly graded

- Caveat: late points and failure-to-do-survey points haven’t been taken away yet

By the way…

- Lecture slides are posted before lecture
- You can follow along on your laptop, if you don’t like looking at the screen
- (Try not to get distracted!)

Inheritance, an overview

- One class can *extend* another.
  - The extending class is called a *subclass*. The extended class is called a *superclass*.
- The subclass *inherits* all the `public` (and `protected`) instance variables and methods of the superclass.

Some classes

```
public class Animal {

   /* All animals deserve to live a good life */
   public void liveAGoodLife() {
      while (true) {
       eat();
       sleep();
      }
   }
   public void eat() {
      System.out.println(“nom”);
   }
   public void sleep() {
      System.out.println(“zzz”);
   }
}
```

Some classes

```
public class Capybara extends Animal {

  @Override
  public void liveAGoodLife() {
      eat();
      sleep();
      swim();
  }
  public void swim() {
    System.out.println(“piddle paddle”);
  }
}
```

A Capybara can still eat and sleep

…but also swims.

Some classes

```
public class Pangolin extends Animal {

  @Override
  public void liveAGoodLife() {
      eat();
      sleep();
      dig();
  }
  public void swim() {
    System.out.println(“dig dig”);
  }
}
```

A Pangolin likes to dig instead

Some classes

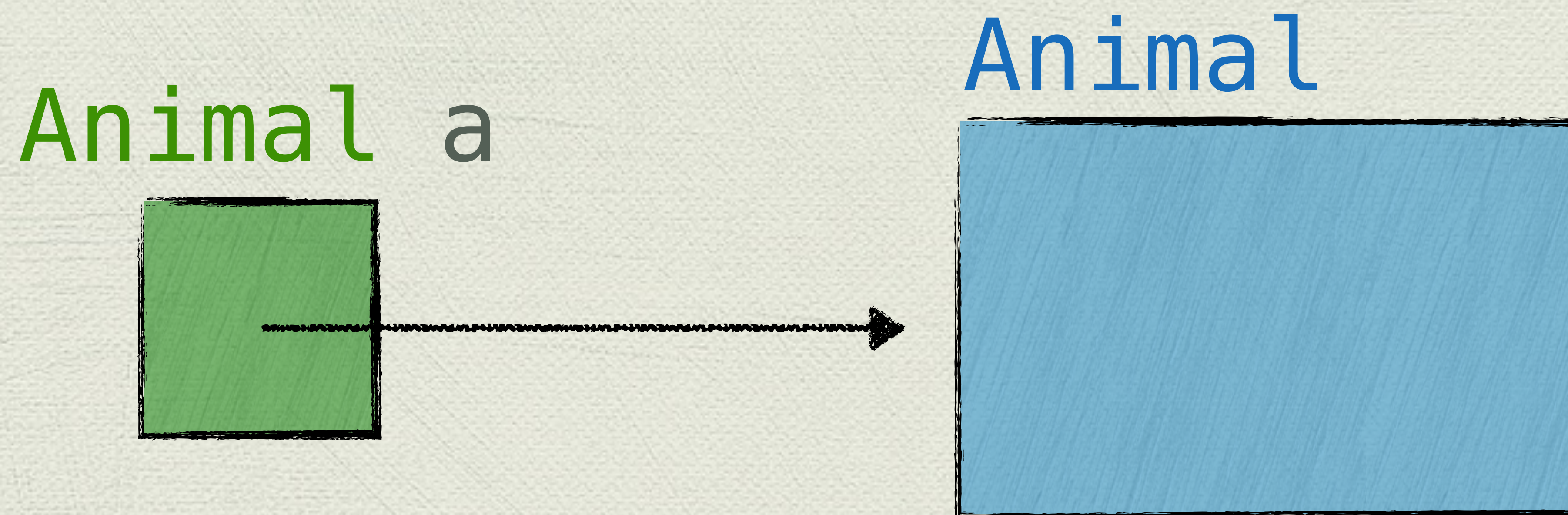
```
public class Wug extends Animal {

  @Override
  public void liveAGoodLife() {
      eat();
      sleep();
      System.out.println("???");
  }
}
```

Does anyone know what a wug is?

- The subclass may *override* (change) some of the methods it inherits.
- But it can't get rid of them.
    - No matter what, the subclass will contain all the methods of the superclass.

Introducing polymorphism

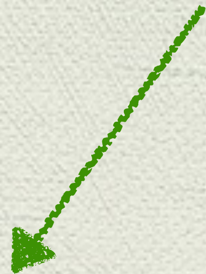
- Earlier, we saw diagrams like this:



- Was it redundant to label the type twice?
- Nope!

Static and dynamic type

Static type



- `Animal a = new Animal();`

Dynamic type

Why “static” and “dynamic”?

- Maybe it’s better for you to think of them as *reference type* and *object type*.

Introducing polymorphism

- The static type must be *the same* as the dynamic type, or some *superclass/interface* of it
  - `Animal a = new Capybara();`
  - `Animal a = new Pangolin();`
- **Polymorphism**, *many forms*, is the idea that one reference can hold different underlying types

```java
public void watchAnimal(Animal a) {
     System.out.println("What a beautiful
     animal! I will watch it forever.”);
    a.liveAGoodLife();
 }
```

- `a` could be a reference to any type of animal: `Capybara`, `Pangolin`, `Wug`…
- Luckily, they are all guaranteed to be able to live a good life. In their own way.

Memories…

- The subclass may *override* (change) some of the methods it inherits.
- But it can't get rid of them.
    - No matter what, the subclass will contain all the methods of the superclass.

```java
public void watchAnimal(Animal a) {
    a.liveAGoodLife();
    System.out.println("What a beautiful animal!");
}
```

- `a` could be a reference to any type of Animal: `Capybara`, `Pangolin`, `Wug`…
- Luckily, they are all guaranteed to be able to live a good life. In their own way.
- So, nothing could possibly go wrong. Polymorphism works!

Why “static” and “dynamic”?

- Slightly non-standard terminology
- But…

Why "static" and "dynamic"?

- **Static type** can be determined from *static analysis*, i.e., without running the code.
- **Dynamic type** cannot be determined for sure until running the code.
  - *Really?*
  - Yes, really.

Strange dynamic type…

```
public class Animal {

    public static void main(String[] args) {
        String userInput = args[0];

        Animal a = null;
        if (userInput.equals("capybara")) {
            a = new Capybara();
        } else if (userInput.equals("pangolin")) {
            a = new Pangolin();
        } else {
            a = new Wug();
        }

        a.liveAGoodLife();
    }
}
```

What is the dynamic type of the object referenced by `a`?

Polymorphism Questions!

- For each of the following questions, discuss what you think will happen with your partner.
- Then we'll see what people think!
- (This is not the actual quiz for this lecture)

What does it print?

```
Capybara c = new Capybara();
c.liveAGoodLife();
```

```
Animal a = new Capybara();
a.liveAGoodLife();
```

What does it print?

```
Animal a = new Capybara();
a.swim();
```

What does it print?

```
Animal a = new Capybara();
Capybara c = a;
c.swim();
```

```
Capybara c = new Capybara();
Animal a = c;
a.liveAGoodLife();
```

Polymorphism Questions!

- Now say we define a class `Person`, and give it the following method

```
public void feedCapybara(Capybara c) {
  c.eat();
}
```

Polymorphism Questions!

- What does the following do?

```
Animal a = new Capybara();
Person p = new Person();
p.feedCapybara(a);
```

```
public void feedCapybara(Capybara c) {
  c.eat();
}
```

Polymorphism Questions!

- Now we add the following methods to `Person`

```
public void observe(Animal a) {
  a.getObservedBy(this);
}
```

```
public void observe(Capybara c) {
  System.out.println(“I love capybaras!”);
}
```

```
public void observe(Pangolin p) {
  System.out.println(“Oh. A pangolin.”);
}
```

Polymorphism Questions!

- And the following method to `Animal`

```
public void getObservedBy(Person p) {
  p.observe(this);
}
```

Polymorphism Questions!

- And the following method to `Capybara`

```
@Override
public void getObservedBy(Person p) {
    p.observe(this);
}
```

Overrides to do the same thing??

Polymorphism Questions!

- What do the following do?

```
Pangolin pang = new Pangolin();
Person p = new Person();
p.observe(pang);
```

```
public void observe(Animal a) {
    a.getObservedBy(this);
}
```

```
public void observe(Pangolin c) {
    System.out.println("Oh. A pangolin.");
}
```

```
public void getObservedBy(Person p){
    p.observe(this);
}
```

Polymorphism Questions!

- What do the following do?

```
Animal a = new Pangolin();
Person p = new Person();
p.observe(a);
```

```
public void observe(Animal a) {
   a.getObservedBy(this);
}
```

```
public void observe(Pangolin c) {
   System.out.println("Oh. A pangolin.");
}
```

```
public void getObservedBy(Person p){
   p.observe(this);
}
```

Polymorphism Questions!

- What does the following do?

```
Animal a = new Capybara();
Person p = new Person();
p.observe(a);
```

```
public void observe(Animal a) {
    a.getObservedBy(this);
}
```

```
public void observe(Capybara c) {
    System.out.println("I love capybaras!");
}
```

In Animal

```
public void getObservedBy(Person p){
    p.observe(this);
}
```

In Capybara

```
@Override
public void
getObservedBy(Person p) {
        p.observe(this);
}
```

Practical polymorphism

- Polymorphism gets hairy in Java
- Lots of rules about what gets called when, what's allowed, etc. See your lab and reading.
- Instead, let's focus on how/why it is actually used

Polymorphism use case 0:
Maintaining abstraction barriers

- It's common to see code like
  - `List l = new ArrayList();`
- *Why?*
  - If it doesn't matter that the list is underlyingly an array, there's no reason to keep that information around
  - Abstraction means *hiding unimportant implementation details*

- Consider the method:

```
  public static void append(ArrayList base,
ArrayList addition) {
    for (Object o : addition) {
      base.add(o);
    }
  }
```

- Why should it have to take in an `ArrayList`? Why not a `LinkedList`? Or some other kind of `List`? It only relies on `List` methods.

- Better:

```
public static void append(List base, List
addition){
   for (Object o : addition) {
       base.add(o);
   }
}
```

- Now someone could use this to process any type of list. Makes sense (for this method), right?

- Bad:

- ```
  public static void append(Object base, Object
  addition){
    for (Object o : addition) {
        base.add(o);
    }
  }
  ```

- Doesn’t even work! `Object` doesn’t have an `add` method, and can’t be iterated through
  ```

Moral of the story

- Make the static type as general as possible, given what you need the object to do
- So you won't accidentally rely on details you shouldn't

Polymorphism use case 1: Extending functionality

- `ArrayList` has some cool methods.
- But you know what method it doesn't have?
  - A *favoriting* method.
  - (I just made this up)

- `FavoritableList` works like an `ArrayList`, except you can favorite an item at a position, and then get it back whenever you want.

- Two additional methods:

```
/* Favorites the item at position i */
public void favorite(int i)

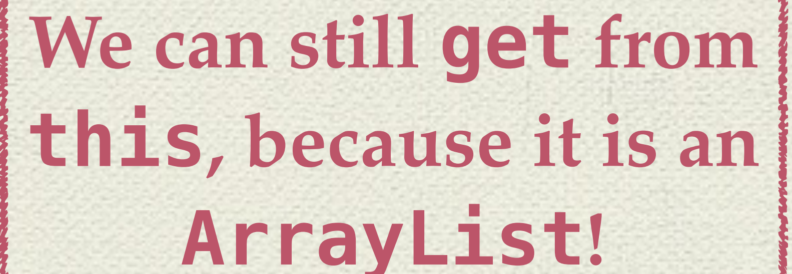
/* Returns the favorite item */
public Object getFavorite()
```

- Here's how we could do it:

```
public class FavoritableList extends ArrayList {
  Object myFavorite;

  /* Favorites the item at position i */
  public void favorite(int i) {
    myFavorite = this.get(i);
  }

  /* Returns the favorite item */
  public Object getFavorite() {
    return myFavorite;
  }
}
```



- `ArrayList` is a pretty useful class.
- But you know how it could be more useful?
- If it was sorted.
- All the time.

- `SortedList` works like an `ArrayList`, except it is always sorted. It only stores integers.
- (We can say it maintains an *invariant* that its items are always sorted)

```
public class SortedList extends ArrayList<Integer> {


   /* Adds the integer to the list in sorted place */
   @Override
   public boolean addd(Integer x) {
      int pos = 0;
      while (pos < this.size() && this.get(pos) < x) {
         pos++;
      }
      super.add(pos, x);
      return true;
   }


/* No matter what, keeps list in sorted order */
   @Override
   public void add(int position, Integer x) {
      this.add(x);
   }
}
```

Changing this method!

Take advantage of the old **ArrayList add** method, to modify **ArrayList**'s private variables

Call our own **add**, which keeps sorted order

Moral of the story

- Use inheritance to add small bits of extra functionality to classes that already exist
- Piggyback off existing functionality

Polymorphism use case 2:
Simplifying code structure

```java
public class Piece {
   private String myType;

   public Piece(String type) {
      myType = type;
   }

   @Override
   public String toString() {
      if (myType.equals("pawn")) {
         return "I'm not important.";
      } else if (myType.equals("bomb")) {
         return "I'm dangerous!";
      } else if (myType.equals("shield")) {
         return "I'm scared.";
      } else {
         return "???";
      }
   }
}
```

A Piece class, like your project. But takes in a type with a String.

Problem: All methods in this class need long conditionals to account for type.

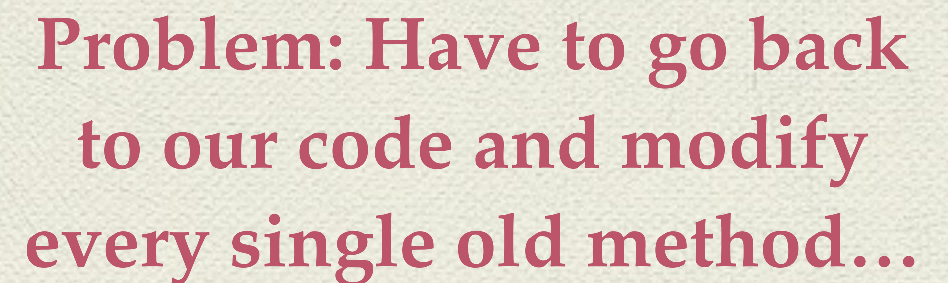
Problem: What if the user input a bad type?

```java
public class Piece {
  private String myType;

  public Piece(String type) {
    if (!(myType.equals("pawn") || myType.equals("bomb")
        || myType.equals("shield")) {
            // fail somehow?
    }
    myType = type;
  }

  @Override
  public String toString() {
    if (myType.equals("pawn")) {
      return "I'm not important.";
    } else if (myType.equals("bomb")) {
      return "I'm dangerous!";
    } else if (myType.equals("shield")) {
      return "I'm scared.";
    } else {
      return "???";
    }
  }
}
```

A solution to invalid type problem? But more conditionals…

- What if we want to add a new type of piece?

```
public class Piece {
  private String myType;

 public Piece(String type) {
  if (!(myType.equals("pawn") || myType.equals("bomb")
   || myType.equals("shield") || myType.equals("knight")){
              // fail somehow?
    }
    myType = type;
  }
  @Override
  public String toString() {
    if (myType.equals("pawn")) {
      return "I'm not important.";
    } else if (myType.equals("bomb")) {
      return "I'm dangerous!";
    } else if (myType.equals("shield")) {
      return "I'm scared.";
    } else if (myType.equals("knight")) {
      return "I'm chivalrous!";
    } else {
      return "???";
    }
  }
}
```



- But there is hope!

```
public abstract class Piece {

}
```

Much simpler logic!

```
public class Pawn extends Piece {
   @Override
   public String toString() {
       return "I'm not important.";
   }
}
```

Impossible to make a `Piece` of a bad type!

```
public class Knight extends Piece {
   @Override
   public String toString() {
       return "I'm chivalrous!”;
   }
}
```

Easy to add a new kind of `Piece`!

Moral of the story

- Using polymorphism can simplify code by eliminating conditionals, making type guarantees, and allowing easier extension
- Tradeoff: Lots of additional separate classes

Conclusion to polymorphism

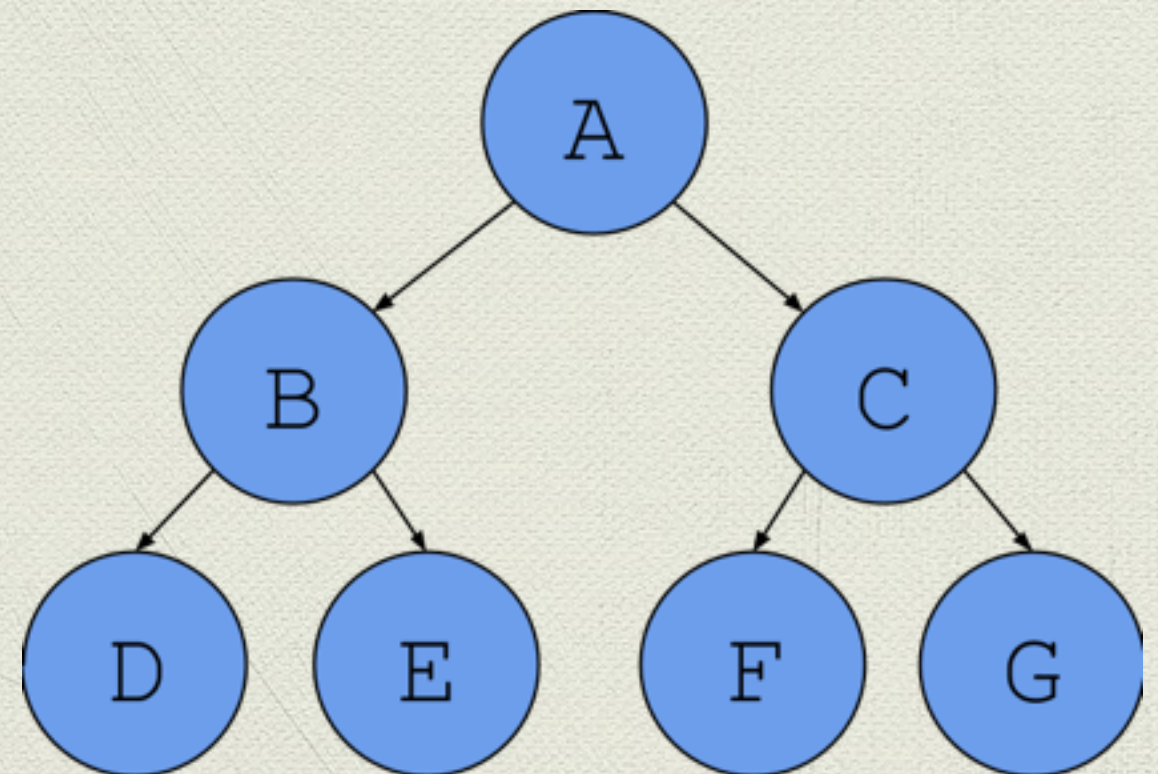
- As you may have found in lab, overuse of polymorphism can lead to confusing code
- But when applied tastefully, it provides several big wins

The best part of lecture!

- Yes!!
- Break!

Cool collections

- In this class, we'll study *data structures* that store collections of data in interesting ways
- Arrays, lists, trees, dictionaries…



Iteration

- When we have a collection of data, usually we want to compute something about it
  - Ex: Given a list of all students in the class, compute their average age
  - Ex: Given a person's family tree, compute a person's ethnic make-up
- This computation will commonly involve looking at each item stored in the collection, one-by-one. This process is called **iteration**

Iterate using a `for` loop

- How to iterate over an array:

```
char[] arr = { 'a', 'b', 'c', 'd' };
int i = 0;
while (i < arr.length) {
  char item = arr[i];
  // do some computation with item
  i++;
}
```

- Would this work for a tree?

Introducing the Incredible Iterator

- We'd like a more *abstract* way of iterating, that would work for any data structure
- Iterating over a tree, or other data structures, could get complicated
- We'll manage the iteration using an *iterator object*.

Iterate using an iterator!

- How to iterate over an array:

```
Array arr = new Array('a', 'b', 'c', 'd' );
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
  // do some computation with item
}
```

- Would this work for a tree?

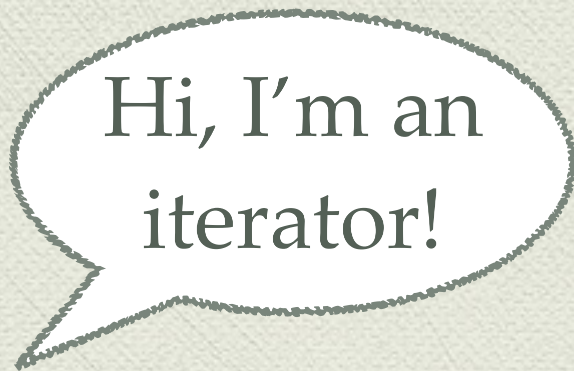
(normal arrays don't actually have a `.iterator()` method. But other collections we find will.)

An object… for iterating?

- Are you serious?
- Yes.

An object… for iterating?

- I like to think of the iterator object as a little insect that crawls along the data structure.



An object… for iterating?

- First, we create a new iterator object. This places down a new ant at the beginning of the array.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

An object… for iterating?

- We ask the ant if it can continue.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Do you have a next?

An object… for iterating?

- We ask the ant if it can continue.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Yes!

An object… for iterating?

- Then we tell it to give us the next.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Well, give me the next, then!

An object… for iterating?

- Then we tell it to give us the next.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

A!

An object… for iterating?

- Then it moves, preparing for another question.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

An object… for iterating?

- We ask the ant if it can continue.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Do you have a next now?

An object… for iterating?

- We ask the ant if it can continue.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Yup!

A	B	C	D	E	F	G

An object… for iterating?

- Then we tell it to give us the next.

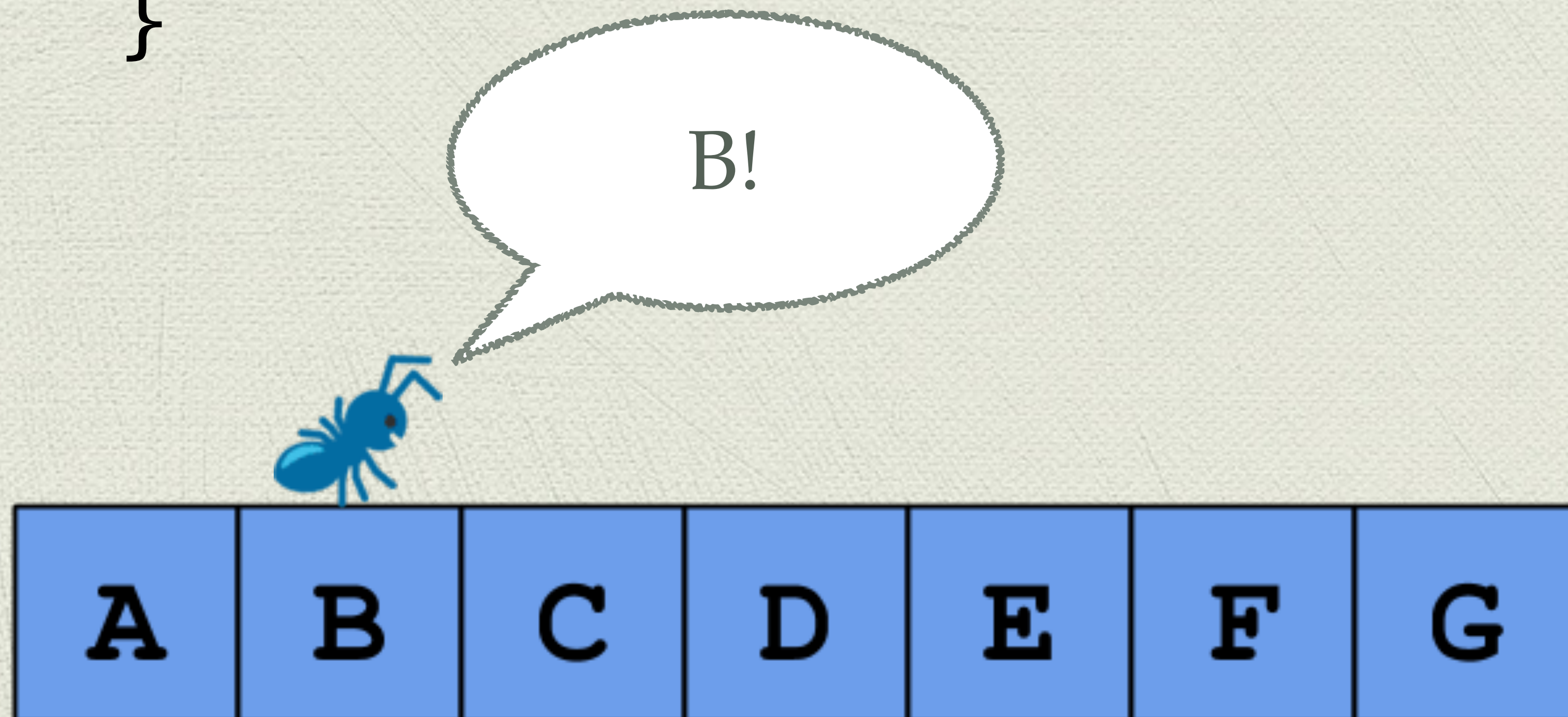
```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

All right then, give me another next!

An object… for iterating?

- Then we tell it to give us the next.

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```



An object… for iterating?

- Then it moves, preparing for another question.

```
Array arr = new Array(‘a','b','c','d','e','f','g');
Iterator iter = arr.iterator();
while (iter.hasNext()) {
  char item = iter.next();
}
```

Notice!!

- When we ask the ant to give us `next`…
- It tells us *what it's currently on*, and then it moves forward to prepare for the next question.
- It *doesn't* move forward, and then tell us what it arrives at

A	B	C	D	E	F	G

Why do we need another object?

- Why not put iteration methods directly in the array class?

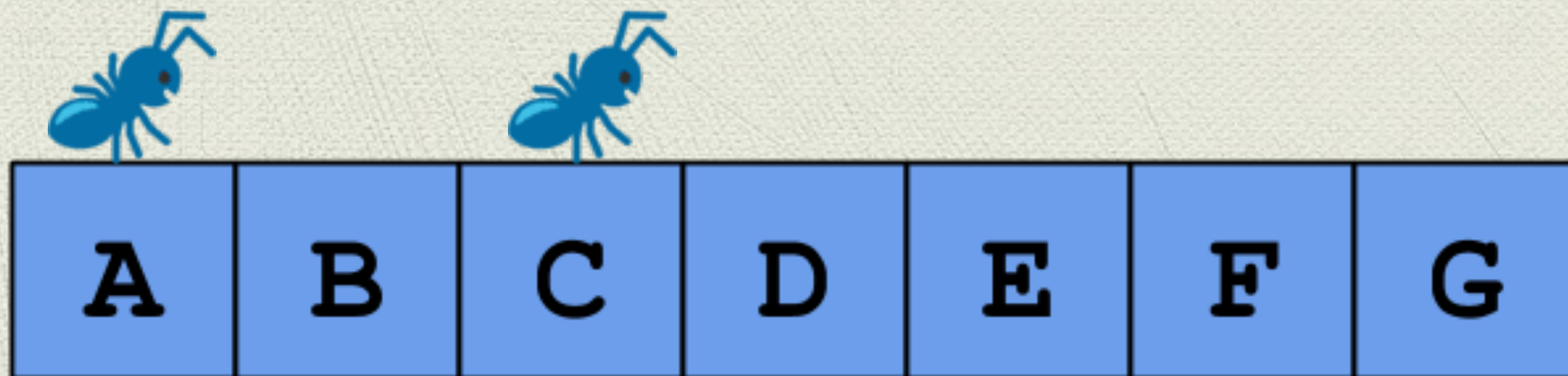
```
Array arr = new Array('a','b','c','d','e','f','g');
arr.initIteration();
while (arr.hasNext()) {
  char item = arr.next();
}
```

An object for iterating!

- We can put down multiple iterators!

```
Array arr = new Array('a','b','c','d','e','f','g');
Iterator iter1 = arr.iterator();
iter1.next();
iter1.next();
Iterator iter2 = arr.iterator();
```

- Each one acts independently.



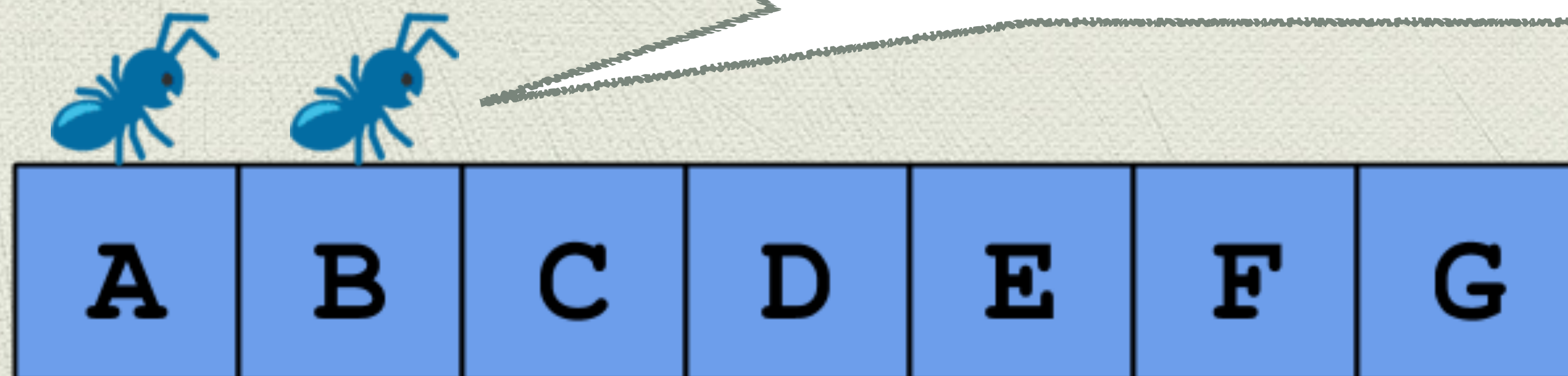
An object for iterating!

- We can define different *types* of iterators

```
Array arr = new Array(‘a','b','c','d','e','f','g');
Iterator iter1 = arr.skipFIterator();
Iterator iter2 = arr.oddIterator();
```

I’ll skip f if I see one.

I only return letters at odd indices!
I don’t even start at index 0…



Questions?

You know what’s coming, right?

Quiz!

- I'd like to introduce a class I made called `Vector` (not Java's `Vector`). It represents a vector from linear algebra (basically an array of numbers)

```
Vector v = new Vector(3, 4, 2, 5);
```

v represents this vector

Quiz!

- `Vector` has exactly one public method, `.iterator()`, which returns a new iterator over the values in the vector.

Quiz!

- Briefly ponder how you could use `.iterator()` to compute the dot product of two vectors.

Dot product

- To compute the dot product of two vectors, multiply corresponding entries of the vectors, then sum the results

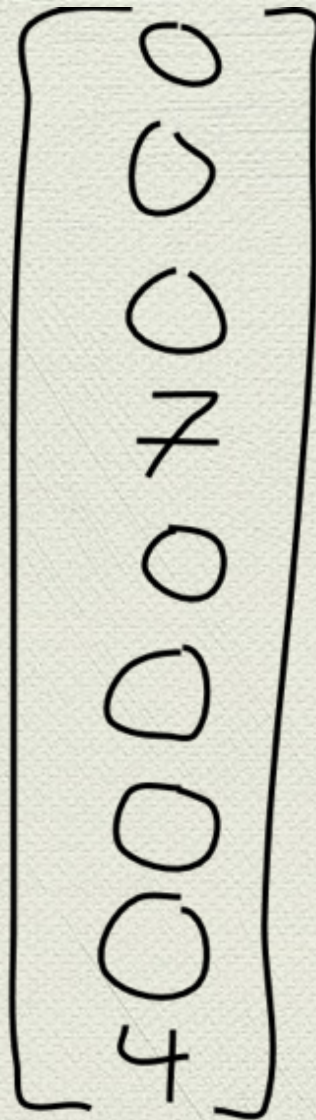
$$\begin{bmatrix} 3 \\ 4 \\ 2 \\ 5 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 3 \\ 2 \\ 3 \end{bmatrix} = 43$$

Quiz!

- The quiz isn't as simple as just computing dot products, however.
- Next, I'd like to introduce a concept known as a *sparse vector*.
- Commonly, while data processing, we have vectors with lots of zeroes…

Quiz!

- A sparse vector.
- Lots of zeroes.



Quiz!

- A sparse vector can be represented more efficiently using two other vectors.
    - One vector records the indices at which the sparse vector is non-zero.
    - The other vector records the values at those positions.

Quiz!

This vector can be represented by the following two:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 7 \\ 0 \\ 0 \\ 0 \\ 0 \\ 4 \end{bmatrix}$$

Indices where non-zero

$$\begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Values at those indices

$$\begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

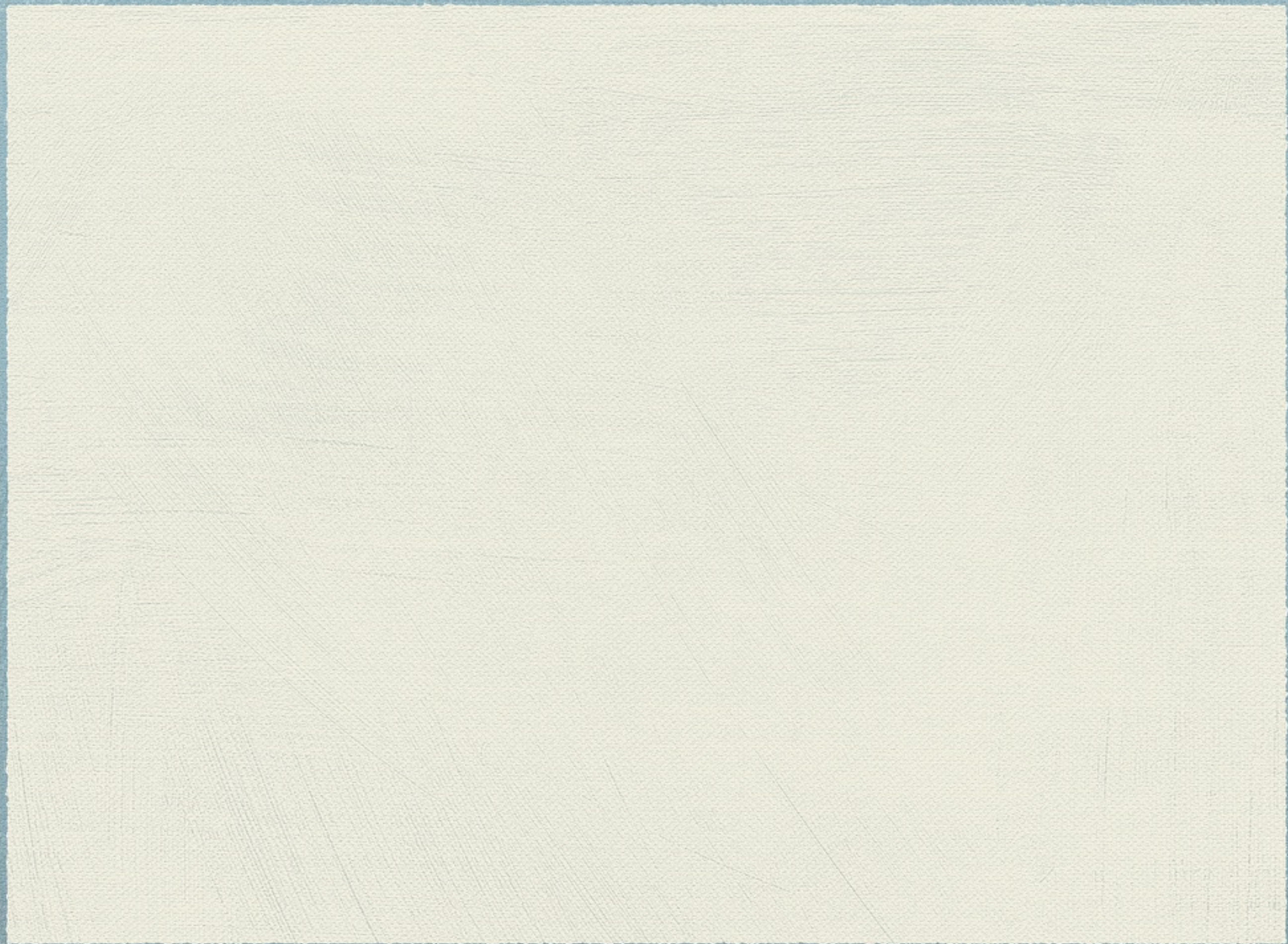
We express the same information using 4 numbers instead of 9!

Quiz!

- Your task: Compute the dot product of one normal vector, and one sparse vector.

```
  public static int dot(Vector x, Vector
yIndices, Vector yValues) {
    // your code here!
  }
```

- Remember: all you have is a `.iterator()` method.



- There were lots of solutions to this problem. Here’s one.

A solution

```java
public static int dot(Vector x, Vector yIndices, Vector yValues) {
   int sum = 0;
   Iterator<Integer> yIndicesIter = yIndices.iterator();
   Iterator<Integer> yValuesIter = yValues.iterator();
   Iterator<Integer> xIter = x.iterator();
   int xIndex = 0;
   while (yIndicesIter.hasNext()) {
      int yIndex = yIndicesIter.next();
      int yValue = yValuesIter.next();
      int xValue = xIter.next();
      xIndex++;
      while (xIndex <= yIndex) {
         xValue = xIter.next();
         xIndex++;
      }
      sum += yValue * xValue;
   }
   return sum;
}
```

- Another proposal…

```
  public static int dot(Vector x, Vector
yIndices, Vector yValues) {
    int sum = 0;
    Iterator xIter = x.iterator();
    Iterator yIter = sparseIterator(yIndices,
yValues);

    while (xIter.hasNext()) {
      int xVal = xIter.next();
      int yVal = yIter.next();
      sum += xVal * yVal;
    }
    return sum;
  }
```

create a new kind of iterator that iterates over the sparse vector *as if it were a normal vector*

- So how do we write an iterator, anyway?
- You’ll see in lab.

Iterator properties

- calling `next` a bunch of times will return each item in the collection *exactly once*
- for some iterators, this is guaranteed to be in a certain order. For others, it's not

Iterator properties (cont.)

- iterating over a collection will **not** modify the collection in any way
- an iteration is not guaranteed to work correctly if the collection is modified while the iteration is taking place

Iterator properties (cont.)

- Ex: The following code is not guaranteed to work, because the list is being modified during the iteration

```
List l = new ArrayList();
// put stuff in l
Iterator iter = l.iterator();
while (iter.hasNext()) {
   int x = iter.next();
   l.add(2);
}
```

add a 2 to the end of the list

Iterator properties (cont.)

- calling `hasNext` will not change anything. Whether you call `hasNext` once or multiple times in a row, the iteration should not change
- `next` should not rely on `hasNext` being called in order to work
- `next` may crash if called too many times