CREATING A LIST

```
import java.util.LinkedList;

public class Zoo {

    public LinkedList<IAnimal> allAnimals;

    public Zoo() {
        this.allAnimals = new LinkedList<>();
    }
```

1 Creates a field called allAnimals that holds elements of type IAnimal:
- The type IAnimal is an interface—all objects that implement the interface (Dillo, Boa, FruitFly, …) can fit in this list, which is what we want for our Zoo!

2 In our constructor, we create the object that represents the list.
(It's also common to write "new LinkedList<IAnimal>()" instead

Here's another way that also shows something else we'll use —our old Zoo had a constructor that used two arguments. A class can have multiple constructors for different ways to set up data.
We can keep this constructor and modify it to *add* the two animals to our new list!

```
public Zoo(IAnimal ani1, IAnimal ani2) {
    this.allAnimals = new LinkedList<>();

    // Add our two animals
    this.allAnimals.add(ani1);
    this.allAnimals.add(ani2);
}
```

3 Java's LinkedList has an add() method to add elements to the list. Here, we use it to add ani1 and ani2

Important concept: in this example, add() modifies the object allAnimals so it contains ani1 and ani2.
In functional programming, it's common to do this by making a new list—instead, Java modifies the existing list!
We'll see this more in the coming weeks.

So how do we use lists?

In the past, we've used recursion. We could do that, but recursion can be messy in Java. Let's learn another common way.

A GENERAL STRUCTURE FOR ITERATION

```java
    // Write a method to determine how many non-normal-size animals are in the zoo
    public int nonNormalCount() {
        int count = 0;

        for (IAnimal ani : this.allAnimals) {

            if (!(ani.isNormalSize())) {
                count = count + 1;
            }

        }

        return count;
    }
```

1 Set up the variable for the result
The starting value (here, 0), will be the result if the list is empty

2 "Iterate over (or "Loop over") all objects in allAnimals:
- Loop runs once per animal in list. The code inside the brackets is called the "body" of the loop
- The name "ani" is added to the environment in the loop—this is often called the "loop variable"
  - The loop variable usually has the same type as the list (IAnimal)
  - On each "iteration" of the loop, ani refers to a different element of allAnimals

3 In the body of the loop, we update something—here, it's "count," the variable we're using to keep track of the result.
(This particular version also could be written as "count += 1" or "count++")

4 When the loop ends, we're (usually) done!

Now we use the return keyword to tell Java what the result of our method is

The type of what we return must match the method's return type, "int"
(from public int nonNormalCount…)

In the normalAverageLength() method, we had written this: (see code example for more)

```
for(IAnimal ani : this.allAnimals) {
    if (ani.isNormalSize()) {
        sum = sum + ani.length
```

ERROR!

This is a problem for two big reasons.

1. Since the class Zoo can access an animal's fields, it can also modify them! In a big system, we usually don't want this, since it means other parts of the code (written by different people) can modify our data!
2. ani has type IAnimal. Not all animals have a length field, only those that extend SizedAnimal! (FruitFly, for example, does not extend SizedAnimal)

Public/Private/Protected fields (Addresses Problem 1)

```
public abstract class SizedAnimal implements IAnimal {
    private int length;
```

- If you mark a field or method as private, it can't be used outside of that class (in this case, SizedAnimal)
  - A private field can ONLY be used within the class where it is declared (ie, SizedAnimal). Not even Dillo or Shark could use it in this form!
- A field or method marked as protected can only be used by the class where it's declared, and its subclasses
  - Marking "length" as protected would allow it to be used by Dillo, Boa, Fish, or anything that extends these (like Shark)
- A field or method marked public can be used by any other class
  - For fields, this means that other parts of the code can modify them!
  - Thus, in Java we usually mark fields as private unless we need them to be public

BEST CHOICE FOR ZOO EXAMPLE.

So how do we give Zoo access to the length field?

As authors of the SizedAnimal class,we get to decide how other classes can access our fields. If we want other classes to be able to get an animal's length, we can write a method like this (in SizedAnimal):

```
public int getLength() {
    return this.length;
}
```

The idea is that we ONLY do this when we decide that a particular field should be exposed

Problem 2: not all animals have getLength()

We just added getLength to SizedAnimal, which provides the method for all animals that extend SizedAnimal.

But in the Zoo class, our list is of type IAnimal. What does Java know about IAnimal?

```
public interface IAnimal {

    // Like a contract that specifies methods that
    // all animals should have

    /**
     *
     * @return true if this animal has a normal size
     */
    public boolean isNormalSize();
```

At this point, all we can do with an IAnimal is call isNormalSize. To use getLength() we need to add it to our interface so that we can do this for all animals. Here's the most important bits, see the full posted code for details.

```
public interface IAnimal {

    // Like a contract that specifies methods that
    // all animals should have

    /**
     *
     * @return true if this animal has a normal size
     */
    public boolean isNormalSize();
    public int getLength();
}
```

Excerpt from averageNormalLength()

```
//...
for(IAnimal ani : this.allAnimals) {
    if (ani.isNormalSize()) {
        sum = sum + ani.getLength();
    }
}
//...
```