Java Collections Framework

- explain the structure of the Java Collections framework.
- program to the generic `Collection` interface including reading and using the APIs
- program to the generic `Iterator` interface including reading and using the APIs
- read and write code that uses a for each loop to iterate over a collection
- determine when a for-each loop can be used and how to avoid concurrent modification of a collection

Reading:

Java Tutorial on Collections:

http://java.sun.com/docs/books/tutorial/collections/index.html

Lessons: Introduction and Interfaces

Review – ArrayLists and Generics

List

- A `List` is an interface defined in the Java libraries.
- An object of type `List` acts like an array except that it automatically grows and shrinks as needed.
- There are several kinds of `List` classes which differ in their performance characteristics
  - `ArrayList, Vector, LinkedList`, etc..
  - Details are described in CPSC 221
  - We will use an `ArrayList` for this lecture

List

- A `List` is an example of a ***generic interface/class***.
- We specify the type of data to be stored in the list when a `List` is declared and instantiated:
  - `List<Account> accts = new ArrayList<Account>();`
    `// a list of Account objects`
  - `List<String> strings = new ArrayList<String>();`
    `// a list of String objects`

List

- The compiler will not allow us to add objects of the wrong type:

```
• List<Account> accts = new ArrayList<Account>();

  accts.add( new Account() );      // OK
  accts.add( new Account() );      // OK
  accts.add( new KitchenSink() );   // won't compile
```

- This is a good thing. The compiler will now check that we're adding the right type of object to our list.

List Methods

- `List` has many useful methods:
  public interface List<E> {

```
...
public boolean add( E item )
   // add at end of list
public boolean add( int i, E item )
   // insert at specific position i
public boolean contains( Object item )
   // is item in the accounts collection
public E get( int i )
   // get item at position i
public E remove( int i )
   // remove account at position i
public int size()
   // gets number of elements in list
   // NOT current capacity of list
...
```

E is a generic parameter

Java Generics

- Note that the `E` in the `List` API is a ***generic parameter*** (or *type parameter)*
- `E` represents the **type** that is specified by the client when the `List` is declared and instantiated
- For example:

```
List<Account> accList;
// E is Account

List<String> strList;
// E is String
```

- For the API for this interface, see the online documentation:

  http://java.sun.com/javase/6/docs/api/index.html

Generic Programming

- *Generic programming* is the creation of programming constructs that can be used with many different types
- A *generic class* has one or more type variables, e.g.
  - public class ArrayList<E>
- These type variables can be instantiated with class or interface types

Arrays and ArrayLists

- Example comparing Arrays and ArrayLists
  - from Head First Java

A simple Animal class hierarchy

```
abstract class Animal {
    void eat() {
        System.out.println("animal eating");
    }
}
public class Dog extends Animal {
void bark() { }
}
public class Cat extends Animal {
void meow() {}
}
```

Arrays

- Let's consider arrays first
- Let's create an array of Animals that hold both cats and dogs
- Let's also create an array of Dogs that can hold only dogs

Arrays

```
public class TestGenerics1 {

public static void main(String[] args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(), new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}

public void takeAnimals(Animal[] animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```

Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}

}
```

Create Animal array

Create Dog array

Call takeAnimals() on each of them

Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```

We can call ONLY the methods declared in type Animal since the parameter is an Animals array

Arrays

```
public class TestGenerics1 {

public static void main(String[] args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(), new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}
```

```
public void takeAnimals(Animal[] animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```

```
>
animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```

ArrayLists

- That was using Arrays
- Let's try the same thing with ArrayLists

ArrayLists

```
import java.util.*;

public class TestGenerics2 {

public static void main(String[]
args) {

new TestGenerics2().go();

}


public void go(){

    ArrayList<Animal> animals = new
ArrayList<Animal>();

    animals.add(new Dog());

    animals.add(new Cat());

    animals.add(new Dog());

takeAnimals(animals);

}
```

```
public void
takeAnimals(ArrayList<Animal>
animals)

{

for(Animal a: animals)

{

a.eat();


}

}

}
```

We've just changed from Animal[] to ArrayList<Animal>
We create an ArrayList of Animals containing Cats and Dogs, and call the takeAnimals() method

ArrayLists

```
import java.util.*;

public class TestGenerics2 {

public static void main(String[]
args) {

new TestGenerics2().go();

}


public void go(){

    ArrayList<Animal> animals = new
ArrayList<Animal>();

    animals.add(new Dog());

    animals.add(new Cat());

    animals.add(new Dog());

takeAnimals(animals);

}
```

```
public void
takeAnimals(ArrayList<Animal>
animals)

{

for(Animal a: animals)

{

a.eat();


}

}

}
```

The method takes an ArrayList<Animal>.
The output is:

```
>
animal eating
animal eating
animal eating
```

ArrayLists

- So far, so good
- With the Array example, we were able to pass a Dog array to a method that took an Animal array parameter
- What happens if we pass an ArrayList<Dog> to our takeAnimals() method, which takes ArrayList<Animal> as a parameter?

ArrayLists

```
public void go(){
    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());
    takeAnimals(dogs);
}
public void takeAnimals(ArrayList<Animal> animals){
        for(Animal a: animals)
        {
            a.eat();
        }
    }
```

ArrayLists

```
public void go(){
    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());
    takeAnimals(dogs);
}
public void takeAnimals(ArrayList<Animal> animals){
        for(Animal a: animals)
        {
            a.eat();
        }
    }
```

```
Exception in thread "main"
java.lang.Error: Unresolved compilation
problem:
        The method
takeAnimals(ArrayList<Animal>) in the
type TestGenerics2 is not applicable for
the arguments (ArrayList<Dog>)

        at
TestGenerics2.go(TestGenerics2.java:13)
        at
TestGenerics2.main(TestGenerics2.java:5
)
```

Assignment with Generics

- Note that List<Dog> is not a subclass of List<Animal>
  - Even though Dog is a subclass of Animal
- Inheritance of type parameters does not lead to inheritance of generic classes
- This restriction saves us some trouble, as just shown

Arrays, ArrayLists, and Polymorphism

- With arrays, we could pass a Dog array to a method expecting an Animal array
  - Polymorphism in action
  - Dog IS-A Animal
- We lost this ability with ArrayLists
- What if we *were* allowed to pass an ArrayList<Dog> to that method? What would happen?
  - Just hypothetically (Java won't let us)

ArrayLists

- What's the worst that could happen?

```
public void takeAnimals(ArrayList<Animal> animals){
animals.add(new Cat()); // bad! A Cat in what should

                        // have been a Dogs-only
                        // ArrayList
```

- So Java just won't let you take this risk
- If you declare a method to take ArrayList<Animal> it can take ONLY an ArrayList<Animal>, not ArrayList<Dog> or ArrayList<Cat>

Arrays and ArrayLists

- So why could we do that with Arrays but not ArrayLists?
  - We could pass a Dog array to a method that takes an Animal array
  - Couldn't somebody add a Cat to the Dog array?
  - Yes! And unfortunately it *would* compile and the error wouldn't be caught until runtime

Runtime

```
takeAnimals(dogs);


public void takeAnimals(Animal[] animals)
{
animals[0] = new Cat();
for(Animal a: animals)
{
a.eat();
}
}
```

```
Exception in thread "main"
java.lang.ArrayStoreException: Cat
	at
TestGenerics1.takeAnimals(TestGenerics1.java:1
9)
	at TestGenerics1.go(TestGenerics1.java:14)
	at TestGenerics1.main(TestGenerics1.java:6)
```

ArrayList

- With ArrayLists, we avoid this nasty problem because type checking occurs when we compile

Motivating Wildcards

- Imagine that we want to add a method to `Bank` that will take a list of accounts and send a directed advertisement to their owners

```
public void spam(List<Account> targetAccounts) ...
```

- We have a problem. We may want to spam a list of `SavingsAccount` but we cannot write:

```
List<SavingsAccount> savingsAccounts
        = new ArrayList<SavingsAccount>();
Bank b = new Bank();

b.spam( savingsAccounts );    //not allowed
```

Bounded Wildcards

- In such cases we can use wildcards in the type parameter:

```
public void spam(
  List<? extends Account> targetAccounts )
{…}
```

- `<? extends Account>` indicates that we can pass a `List` of any type that is a subtype of `Account`
- So we can now pass a `List` of `Account` or `SavingsAccount` or any other type that's a subtype of `Account`.

Bounded Wildcards - Question

- When we use a bounded wildcard, we can visit the items in the collection but we are not allowed to add an item to the collection.

```
public void spam(List<? extends Account>
targetAccounts )
{
   targetAccounts.add( new Account() );
   //…
}
```

- Why is this not allowed?

Bounded Wildcards - Question

- We can answer that by revisiting our Animals/Dogs/Cats example
- We discovered that we could not pass ArrayList<Dog> to a method expecting an ArrayList<Animal> parameter
- But now we know about a workaround: bounded wildcards

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```

Now we can pass in an ArrayList<Dog> or ArrayList<Cat>

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```

But what's the difference? Don't we have the same problem as before? This allows us to pass in an ArrayList<Dog> but somebody could still add a Cat to the ArrayList of Dogs, right?

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```

But what's the difference? Don't we have the same problem as before? This allows us to pass in an ArrayList<Dog> but somebody could still add a Cat to the ArrayList of Dogs, right?

No! When you use a bounded wildcard in a method parameter, the compiler will not let you add anything to that list. You can use the list but not add anything to it. Problem solved.

Java Collections Framework

- We have examined the use of one collection class, `ArrayList`, and observed that we sometimes need other classes that support very similar operations (with some differences).
- We will now see how Java uses a hierarchy of interfaces to abstract the common behaviours that are shared by these classes.
- This hierarchy is called the ***Java collections framework***.

Java Collection Framework

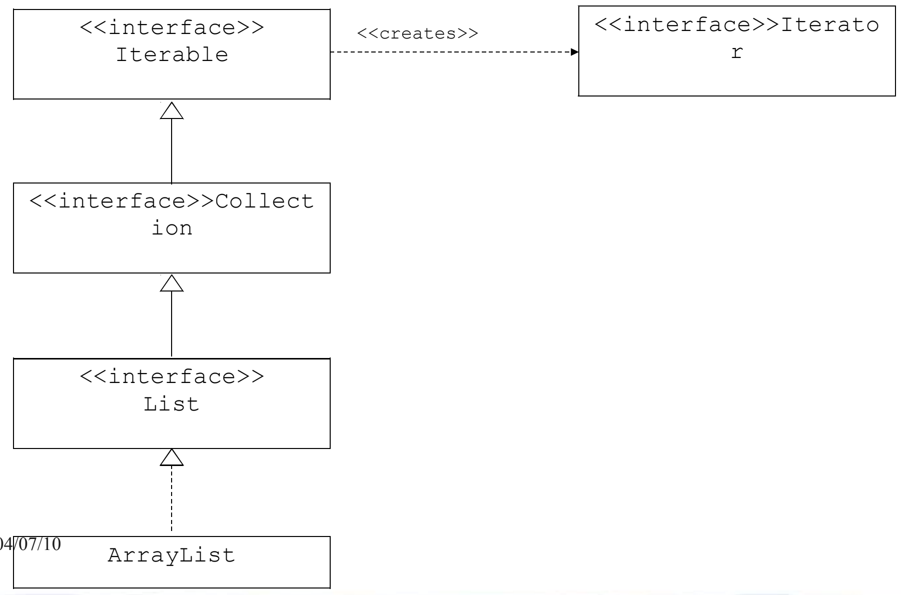
- The Collections Framework is in the `java.util` package.
- The interfaces and classes in this package provide
  - standardized interfaces with multiple implementations of most data structures (e.g., `List, Set, etc.`)
  - efficient, highly-optimized implementations of common data structures (e.g., `ArrayList`)
  - interoperability between programs by making it easier to exchange collections

Java Collection Framework

Consists of 3 components:

- Interfaces
  - provide specifications for the behaviour of the collections
  - form inheritance hierarchies
- Implementations
  - provide specific structures that store the elements and relevant operations on those structures
  - each interface may have multiple implementations that differ only by which optional operations they implement and by the efficiency of the operations
- Algorithms
  - polymorphic algorithms that manipulate data stored in collections
  - are not members of any collection

Some Collection Interfaces



Collection Interfaces

- The `Collection` interface specifies methods that are applicable to all collections (lists, sets and queues – more later).
- The `List` interface specifies methods that are particular to lists (e.g., the ability to add an element at a specific location in the list).

Iterable Objects

- The `Iterable` interface has a single method and is defined in `java.lang` as:

```
public interface Iterable<T>
{
     // Returns an iterator over a set of
     // elements of type T.
    Iterator<T> iterator();
}
```

- Each iterable object can return an `Iterator`:
  - An iterator is an object that allows us to visit the items in a collection
- This is another example of a generic type. The type `T` is a generic type that will be specified when the iterator is declared and instantiated.

The **`Iterator`** Interface

- Is defined in `java.util` as:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();  // Optional
}
```

- `hasNext()` returns true if there is another element to visit
- `next()` returns the next object in the collection **and** advances the iterator to another object that has not been visited
- `remove()` removes the object that was returned by the last `next()` operation
  - can be called only once per call to `next()`
  - otherwise `IllegalStateException` is thrown.

The **`Iterator`** Interface (cont'd)

- Some notes on the *optional* `remove()` method:
- Methods in an interface that are documented to be optional:
  - provide flexibility
  - allow for a reduction in the number of interfaces needed
  - **must** be implemented by classes that implement the interface although those implementations may do nothing more than throw an `UnsupportedOperationException.`

The **Collection** Interface

```
public interface Collection<E> extends Iterable<E>  {
   int size();
   boolean isEmpty();
   boolean contains(Object o);
   boolean add(E o);           // Optional
   boolean remove(Object o);      // Optional
   Iterator<E> iterator();
   // Bulk Operations
   boolean containsAll(Collection<?> c);
      ... more ...
   // Array Operations
   Object[] toArray();
   <T> T[] toArray(T[] a);
   // Object operations; allow collections to customize
   boolean equals(Object o);
   int hashCode();
}
```

The **Collection** Interface (cont'd)

- Provides a general set of methods applicable to all collections
- Used as a base for more specific sub-interfaces (e.g. `List` and `Set`)
- **Note**: the `contains()` method uses `equals()` for comparison.

Collection-Iterator - Example

- Complete the following method that prints out all the elements in a collection of strings:

```
public static void print( Collection<String> col )
{
```

Collection-Iterator – Generic Example

- Complete the following method that prints out all the elements in a collection of any type:

```
public static <T> void print( Collection<T> col )
{
```

Tea break!

For-Each Loop

- Java provides a special type of for loop (called ***for-each loop***) which can be used with any collection
- Example: Another version of the method that prints out all the elements in a collection of strings:

```
public static void print( Collection<String> col )
{
    for(String str : col)
   {
        System.out.println(str);
   }
}
```

- Write the generic version using a for-each loop

Generic Version

```
public static <T> void print( Collection<T> col )
{
     for(T str : col)
    {
        System.out.println(str);
    }
}
```

T vs. ?

- We learned about bounded wildcards
  - ? extends B (any subtype of B)
  - ? super B (any supertype of B)
  - ? (any type)
- Couldn't we just use ? in this case?

T vs. ?

- T allows the user to pass a Collection of any type
  - T gets instantiated as that type
- ? also allows any type
- But how would we define the method?

T vs. ?

```
public static void print( Collection<?> col )
{
    for(  str : col)
   {
       System.out.println(str);
   }
}
```

What goes here? Object?

Generic Parameters

- The advantage of generic parameters like T is that T gets instantiated with whatever type the user supplies, so that all instances of T are essentially replaced with that type

Generic Version

So this...

```
public static <T> void print( Collection<T> col )
{
    for(T str : col)
    {
        System.out.println(str);
    }
}
```

essentially becomes this...

```
public static void print( Collection<String> col )
{
    for(String str : col)
    {
        System.out.println(str);
    }
}
```

For-Each Loop and Collection Modification

- A for-each loop cannot modify the collection over which the loop iterates
  - if this rule is violated Java throws a `ConcurrentModificationException`
- The following method that removes all accounts with low balance IS WRONG:

```
public static void removeBelow(
    Collection<Account> accounts, double limit ) {
  for ( Account acc : accounts ) {
      if ( acc.getBalance() < limit )
         accounts.remove(acc);    // WRONG
  }
}
```

Iterators and Collection Modification

- A collection cannot be modified during the time an iterator iterated over it, unles it is done through the iterator `remove()` method
  - if this rule is violated Java throws a `ConcurrentModificationException`
- The following method IS WRONG:

```
public static void removeBelow(
    Collection<Account> accounts, double limit ) {

    Iterator<Account> itr = accounts.iterator();
    while( itr.hasNext() ){
         Account acc = itr.next();
         if( acc.getBalance() < threshold )
             accounts.remove(acc);    // WRONG
    }
}
```

Iterators and Collection Modification (cont'd)

- The following is the correct code for this method:

```
public static void removeBelow(
    Collection<Account> accounts, double limit ) {

    Iterator<Account> itr = accounts.iterator();
     while( itr.hasNext() ){
          Account acc = itr.next();
          if( acc.getBalance() < threshold )
              ______________________
    }
}
```

Iterators

- We've now seen a few ways we can iterate over an ArrayList
  - for-each loop (enhanced for loop)
  - while loop coupled with get() method and index
  - Iterator

for-each loop

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");
for (String s: myArr)
{
  System.out.println(s);
}
```

for-each loop

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");
for (String s: myArr)
{
  System.out.println(s);
}
```

```
>
hello
world
```

Using while and get()

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

int x = 0;
while (x < myArr.size()){
String g = myArr.get(x);
System.out.println(g);
x++;
}
```

Using while and get()

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

int x = 0;
while (x < myArr.size()){
String g = myArr.get(x);
System.out.println(g);
x++;
}
```

```
>
hello
world
```

Using an Iterator

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

Iterator<String> it = myArr.iterator();
while (it.hasNext())
{
String s = it.next();
System.out.println(s);
}
```

Using an Iterator

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

Iterator<String> it = myArr.iterator();
while (it.hasNext()) ◄──── Check if there is another item
{
String s = it.next(); ◄──── Get that item
System.out.println(s);
}
```

Using an Iterator

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

Iterator<String> it = myArr.iterator();
while (it.hasNext())
{
String s = it.next();
System.out.println(s);
}
```

Which should I use?

- It depends
- We've just seen one case where you cannot use a `for:each` loop

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");

for ( String s : myArr ) {
if (s.equals("world"))
{
   myArr.remove(s);
   // won't work
}}
```

```
Exception in thread "main"
java.util.ConcurrentModificationException
        at
java.util.AbstractList$Itr.checkForComodificatio
n(AbstractList.java:372)
        at
java.util.AbstractList$Itr.next(AbstractList.java:3
43)
        at
WaysToIterate.main(WaysToIterate.java:16)
```

Instead...

```
ArrayList<String> myArr = new ArrayList<String>();
myArr.add("hello");
myArr.add("world");
Iterator<String> it = myArr.iterator();
while (it.hasNext()){
  String s = it.next();
  if (s.equals("world")){
    it.remove();
     }
  }
```

When to use for-each

- The for-each loop is useful when you want to iterate over an *entire* collection (rather than partway) and you don't plan to modify it
- There are many situations with collections where a for-each loop is extremely inefficient or impossible

ArrayList and Iterators example

- Let's write a method that can take an ArrayList<Animal>, remove any Dogs in the list, add each Dog to a new ArrayList<Dog> and return the ArrayList<Dog>
- We'll make it a static method

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){


}
```

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){


}
```

Takes an ArrayList<Animal>

We will return an ArrayList<Dog>

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){
  ArrayList<Dog> dogList = new ArrayList<Dog>();
  Iterator<Animal> it = animList.iterator();


}
```

We opted for an Iterator here. Could we use a for-each loop instead?

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){

  ArrayList<Dog> dogList = new ArrayList<Dog>();

  Iterator<Animal> it = animList.iterator();

  while (it.hasNext()){

    Animal a = it.next();


}
```

With an Iterator, we first check whether there is a next item, and if so we move to that item

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){

  ArrayList<Dog> dogList = new ArrayList<Dog>();

  Iterator<Animal> it = animList.iterator();

  while (it.hasNext()){

    Animal a = it.next();

    if (a instanceof Dog){


    }

  }

return dogList;

}
```

Check if this Animal is a Dog (or a subclass of Dog)

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){
  ArrayList<Dog> dogList = new ArrayList<Dog>();
  Iterator<Animal> it = animList.iterator();
  while (it.hasNext()){
    Animal a = it.next();
    if (a instanceof Dog){
      it.remove();


    }
  }
}
```

Call the Iterator remove() method. Since we are modifying the ArrayList, we had to use an Iterator and its remove() method. Could not have used a for-each loop.

Dog Filter

```
public static ArrayList<Dog> dogFilter(ArrayList<Animal>
animList){
  ArrayList<Dog> dogList = new ArrayList<Dog>();
  Iterator<Animal> it = animList.iterator();
  while (it.hasNext()){
    Animal a = it.next();
    if (a instanceof Dog){
      it.remove();
      Dog d = (Dog) a;
      dogList.add(d);
    }
  }
return dogList;
}
```

Cast the Animal object to be a Dog, and add it to the ArrayList<Dog>.

Finally, return the ArrayList<Dog>

Dog Filter Example

```
public static void main(String[] args) {
  ArrayList<Animal> alist = new ArrayList<Animal>();
  alist.add(new Dog());
  alist.add(new Cat());
  alist.add(new Dog());
  ArrayList<Dog> doglist = dogFilter(alist);
  System.out.println(alist.size());
  System.out.println(doglist.size());
```

What gets printed?

```
}
```

Dog Filter Example

```
public static void main(String[] args) {
  ArrayList<Animal> alist = new ArrayList<Animal>();
  alist.add(new Dog());
  alist.add(new Cat());
  alist.add(new Dog());
  ArrayList<Dog> doglist = dogFilter(alist);
  System.out.println(alist.size());
  System.out.println(doglist.size());
```

```
>
1
2
```

```
}
```

In-Class Exercise II

1. Write a public static method that accepts a collection of type `Collection<String>` as an argument and removes all objects in collection `c` that satisfy the test: `boolean test(String)`

2. Write a generic public static method that removes duplicates from a collection.

Learning Goals Review

- explain the structure of the Java Collections framework.
- program to the generic `Collection` interface including reading and using the APIs
- program to the generic `Iterator` interface including reading and using the APIs
- read and write code that uses a for each loop to iterate over a collection
- determine when a for-each loop can be used and how to avoid concurrent modification of a collection