CS 228: Introduction to Data Structures
Lecture 17
Wednesday, October 5, 2016

`AbstractCollection<E>`

Implementing the `Collection` interface from scratch would be daunting — just look at the number of methods specified in the javadoc! Fortunately, Java offers us `AbstractCollection<E>`, a generic abstract class that implements all the methods of `Collection<E>`, except `size()` and `iterator()`.

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

   public abstract Iterator<E> iterator();

   public abstract int size();
```

All other methods are implemented based on `iterator()` and `size()`. For example, here are `isEmpty()` (which is true if and only if the collection is empty) and `contains()`.

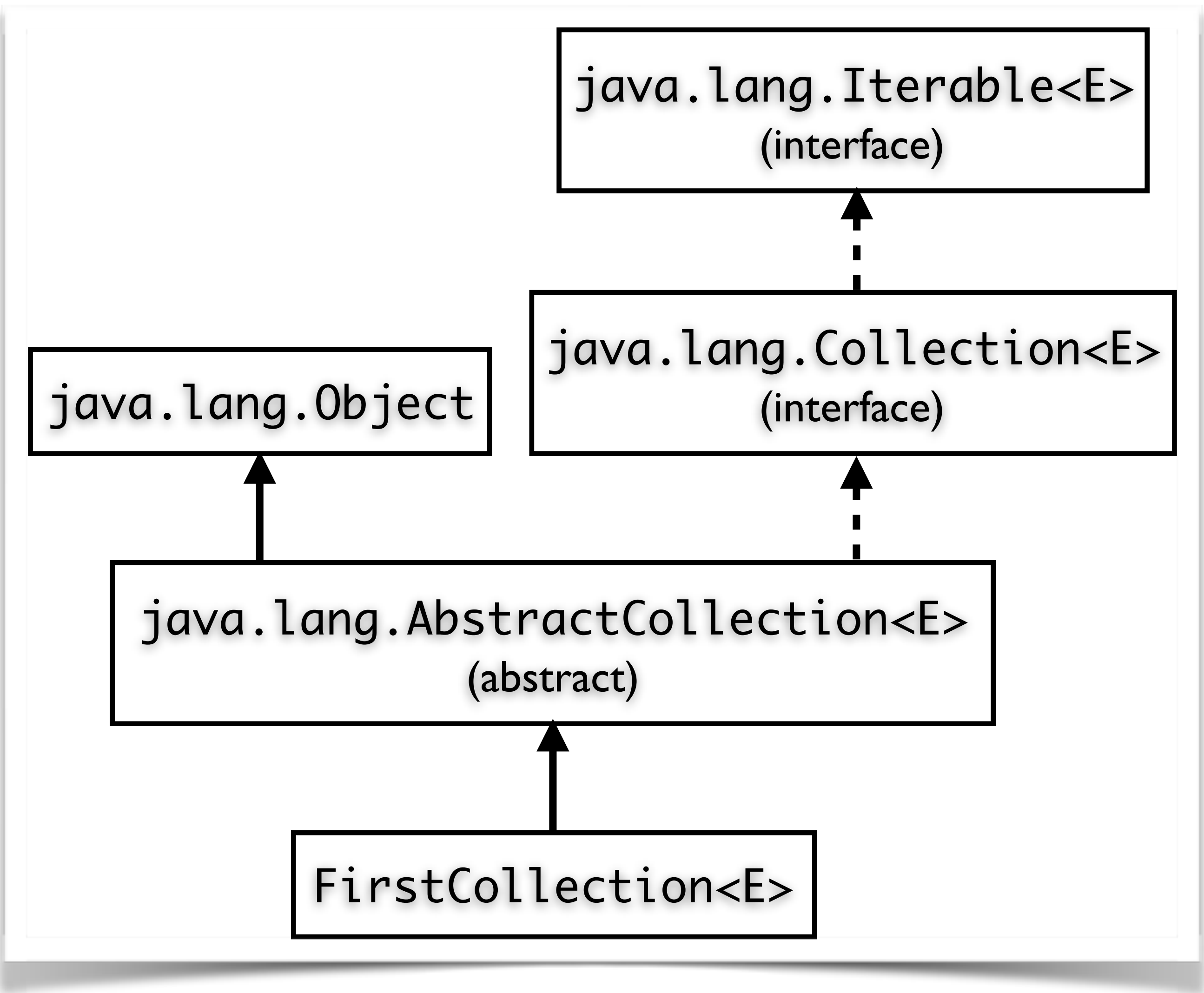
```java
public boolean isEmpty() {
   return size() == 0;
}

public boolean contains(Object o) {
   Iterator<E> e = iterator();
   if (o==null) {
      while (e.hasNext())
        if (e.next()==null)
           return true;
   } else {
      while (e.hasNext())
        if (o.equals(e.next()))
           return true;
   }
   return false;
}
```

Some methods of `Collection` are *optional*; i.e., they are not required to be implemented by an implementing class. Optional methods in `AbstractCollection` are implemented in a simple fashion: Just throw an `UnsupportedOperationException`, like this:

```java
public boolean add(E o)
{
   throw new UnsupportedOperationException;
}
```

`AbstractCollection` serves as a starting point for concrete implementations of `Collection`.
The figure on the next page shows the portion of the Java class hierarchy that we will study in the next few weeks.
The figure also shows where `FirstCollection`, our first implementation of the `Collection` interface, lies within that hierarchy.

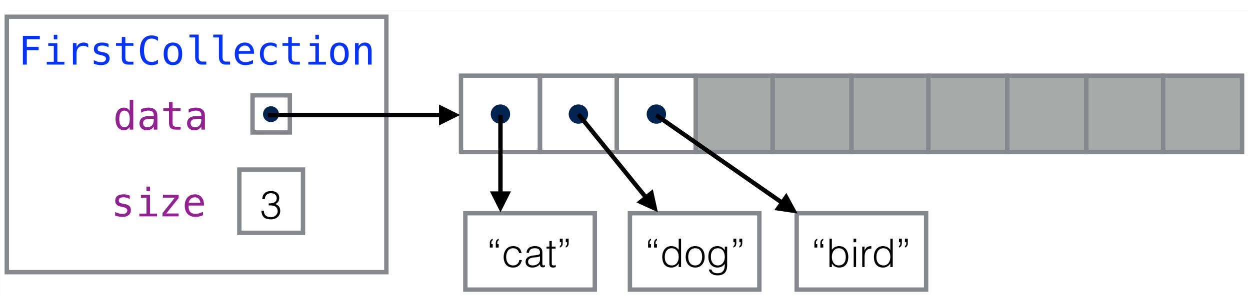


An Array-Based Generic Collection

We start with a simple array-based implementation of `Collection<E>` called `FirstCollection<E>`. The code is posted on Blackboard. We will only discuss in detail a few representative methods. You are responsible for carefully reading the posted code.

Basic Structure

The basic elements of `FirstCollection` are a `data` array, which stores the items, and a `size` field, which indicates how many slots of `data` are being used.



The class definition begins like this:

```
public class FirstCollection<E> extends
   AbstractCollection<E>
{
  private static final int DEFAULT_SIZE
       = 10;
  private E[] data;
  private int size;
```

There are two constructors. One takes an `initialCapacity` argument, specifying the initial length of *data*. The default constructor initializes `data` to `DEFAULT_SIZE` (= 10).

```
  public FirstCollection()
  {
    this(DEFAULT_SIZE);
  }

  public FirstCollection
     (int initialCapacity)
  {
    data
      = (E[]) new Object[initialCapacity];
    size = 0;
  }
```

To complete the description of `FirstCollection`, we need to explain how to implement `add()`, `size()`, and `iterator()`.

`add()`

Adding is easy: just put the new item in the next available slot at the end of the `data` array. But what if we run out of space? The code below handles this possibility by invoking a `checkCapacity()` method, which, if necessary, doubles the capacity of the `data` array to accommodate the new item.

```
public boolean add(E item)
{
  checkCapacity();
  data[size++] = item;
  return true;
}
```

`checkCapacity()` ensures that there is enough space for a new item. If there isn't, it copies the entire array into a new array double the size of the original one.

```
private void checkCapacity()
{
  if (size == data.length)
  {
    data = Arrays.copyOf(data,
                              data.length * 2);
  }
}
```

An `add()` takes O(1) time if `data` is not full. If `data` is full, though, it takes O(n) time, where n = `data.length`. Which is more typical?

Notice that the good case occurs much more frequently than the bad case: In a sequence of 161 `adds`, starting with the default size of 10, the bad case occurs only 5 times: at operations 11, 21, 41, 81, and 161. As the array gets bigger, the time between bad cases increases drastically — the next bad add occurs at operation 321!

Theorem. The **total time** for a sequence of n add operations is O(n).

Proof: Suppose we start out at size 1 and that n is a multiple of 2, say n is 2^p. Each time we run out of space, we double the capacity. Then, there are resize operations for sizes 1, 2, 4, 8, 16, ..., 2^{p-1}. A resize operation on an array of size k takes time O(k). So the total cost of all the resize operations is

$$O(1 + 2 + 4 + ... + 2^{p-1}) = O(2^p) = O(n).$$

The theorem implies that the time for n `add()` operations averages out to O(1) per operation. More formally, we say that the ***amortized time*** for `add()` is O(1).