COP 3330 – Object Oriented Programming in Java
Classes – HAS-A Relationship

Contact, AddressBook Classes

This is the first lecture about a relationship between classes. In our previous programs, we've effectively seen one distinct type of object only. In this example, we'll see a program with 2 types of objects (and three classes total). More specifically, the two classes will be related to each other in a specific way: an AddressBook object will contain multiple Contact object. A common way to say this is: "an AddressBook object has a Contact object." Notice that the "has a" relationship isn't necessarily unique to an individual class. It's possible that an object of class A has an object of class B and an object of class C. (A human has two arms and a head and quite a few other things.)

Contact Class

A contact, for the purposes of this example is information about a person that you keep. To keep things simple, our Contact object will store the following instance variables:

```
private String name;
private int age;
private long phonenumber;
private int bday;
```

One thing that's interesting here is that we store the birthday as a single integer. The user of this class need not know this. Here is the constructor:

```
public Contact(String n, int a, long p, int month, int day) {
    name = n;
    age = a;
    phonenumber = p;
    bday = 100*month + day;
}
```

Thus, we store two integers, both guaranteed to be less than 100 as one integer, guaranteed to be less than 10,000. (Technically since month ≤ 12 and day ≤ 31, the maximum the integer bday could be set to is 1231.) If we ever need to extract each individual value, we can simply do so via integer division and mod by 100. In fact, there's nothing special about 100 here, we could have used any integer 32 or greater so long as we don't overflow int.

Also notice that the phone number is stored as a long. Funny story, when I first created this example in 1998 or 1999, phone numbers were 7 digits long. Thus, they easily fit in an integer, because numerically, all phone numbers back then were less than 10 million. However, by just adding an area code, the maximum numeric value of a phone number can be upto 10 billion. Unfortunately, the storage of an integer only goes to roughly 2 billion. (As previously discussed, the actual upper limit of what an int can store is $2^{31} - 1$.)

Of course, we've used 10 digit dialing for quite a few years now, so in the middle of typing up these notes, I was thinking about actually typing in a real phone number and before I even started I realized that I'd overflow int, and then I went back and changed the example.

The rest of the class just manages storing a single Contact entry. Here are the names of the methods in the class with a brief description of what they do:

```
// Changes the phone number of this contact to newnum.
public void changeNumber(int newnum);

// Implements the passing of Contact's birthday.
public void Birthday();

// Returns a string representation of this Contact.
public String toString();
```

The rest of the methods in the class are what we call accessors: they allow us to access (but not change) instance variables of the class. (For example, getAge() returns the age of this Contact.)

A single Contact object doesn't seem too useful on its own. What would be more useful is if you could store several Contact objects, much like an AddressBook one might keep. Eventually we'll develop three AddressBook classes. In this lecture we'll see two of them that will employ slightly different functionality:

1. AddressBook
2. AddressBook3

Both will use arrays in Java. (AddressBook2 uses an ArrayList which hasn't been taught yet.)

The first AddressBook will limit the total number of Contacts to 10. This design decision was made simply to make the code of the class as simple as possible for the first time students will be formally learning the HAS-A relationship.

The second example still uses an array, but if the array gets full, it resizes that array to be larger so that more Contact objects can be stored.

Here are the instance variables of the Contact class:

```
private Contact[] friends;
private int numfriends;
```

The array, at any given time will have anywhere in between 0 and 10 Contact objects. But, for the entirety of the life of the Contact object, the size of the array friends will be 10. This is why we need a second instance variable, numfriends. (When looping through our valid Contacts, we can't just loop upto friends.length, exclusive.)

We only provide a default constructor which creates an AddressBook, initially with 0 contacts:

```
public AddressBook() {
    friends = new Contact[10];
    numfriends = 0;
}
```

Here is a list of methods available in the AddressBook class:

```
// Returns true iff we can add a friend.
public boolean canAdd();

// Adds the contact c to this AddressBook. Returns true if successful, false
// if the AddressBook was already full and c wasn't added.
public boolean addContact(Contact c);

// Returns a String represetation of this AddressBook.
public String toString();

// Returns the number of friends currently in this AddressBook
public int numContacts();

// Returns the first Contact object in this AddressBook with a name
// equal to s. If no such Contact object exists, null is returned.
public Contact getContact(String s);

// Attempts to delete a Contact in this AddressBook with the name s.
// Returns true if a deletion was made, false if no such Contact was
// found.
public boolean deleteContact(String s);
```

We need one private method to help us out for the last task:

```
// Returns the index of the first Contact in this AddressBook with the
// name equal to s, or -1 if no such Contact exists.
private int haveContact(String s);
```

In AddressBook (version one), the addContact method is useful since there's a clear situation when we can't add a Contact. (For our version where we make a larger array, we just have this method return true. For a practical application, we would only not return true if we couldn't allocate an array of the appropriate size. But this would require catching an Exception, which we haven't learned yet.)

Here's the implementation of the canAdd method:

```
public boolean canAdd() {
    return numfriends < 10;
}
```

Now, let's take a look at the implementation of addContact. Our strategy will simply be to add a Contact after the last spot in the array that has an actual Contact stored. This method will return false and do nothing else if the array is full, which the above method checks, so we can use it.

```
public boolean addContact(Contact c) {

    if (!canAdd()) return false;

    friends[numfriends] = c;
    numfriends++;
    return true;
}
```

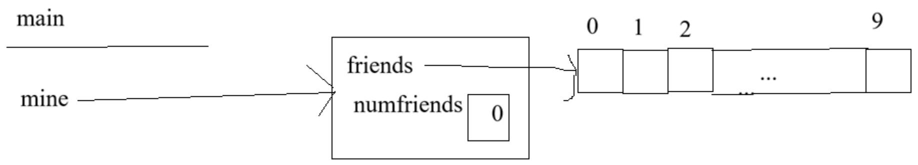
So the first line screens out cases where we can't add a contact.

All we have to do to add a contact is store the reference to the contact object in our array (of references), update our numfriends counter (of this object) to indicate that we have one more friend that before and return true.

Imagine the following lines of code in a main method from another class:

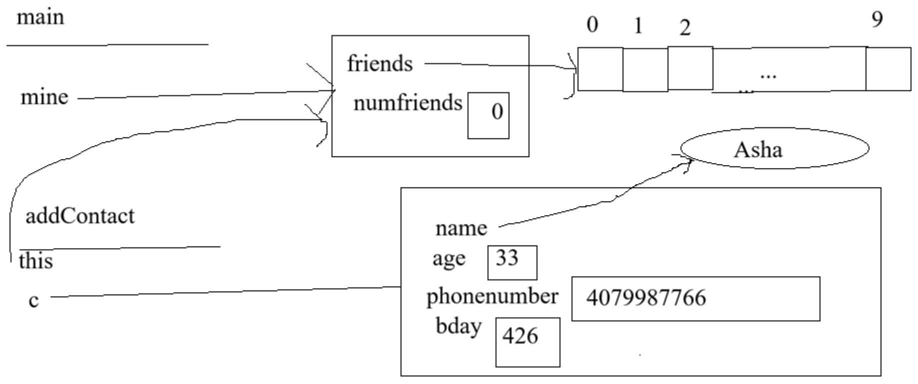
```
AddressBook mine = new AddressBook();
mine.addContact(new Contact("Asha", 33, 4079987766, 4, 26));
```

The picture after the first line of code is:

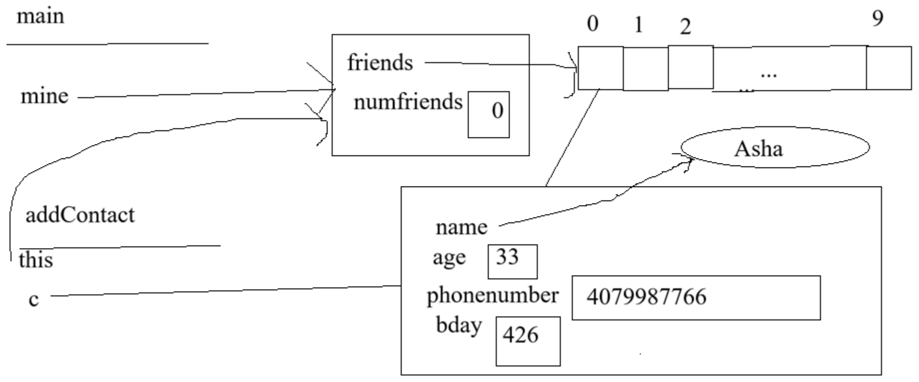


Each of the array references will default to null (just like an array of integers defaults to 0.)

Now, the Contact constructor will fully run, create an object (using the information in the actual parameter list) and return a reference to that object. THAT reference then gets sent to the addContact method, so technically, that reference never has a name in the code because I am writing two actions in the same line of code. None the less, the formal parameter c will automatically reference this object right when the method starts:

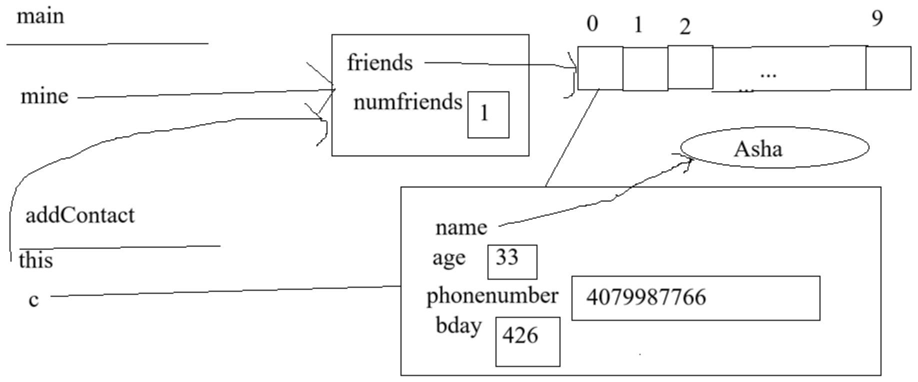


Once the method runs, it will assign friends[numfriends] of this object to c:



Thus, whenever we assign a reference to another reference, as we've seen before, the reference on the left-hand side will just point to the object that the reference on the right-hand side is already pointing to.

Finally, so that future code works, we must keep our object consistent. Since we now have one friend, we have to add one to the numfriends component of this object:



What we can see from this detailed picture is that if you write each method to properly operate on the object to complete whatever task is at hand, then calling the method becomes very easy and the programmer calling the methods provided doesn't need to worry themselves with any of these details. Thus, there are two views when using classes when programming: the view of building the class where you DO need to understand the details of how the object is stored and how to algorithmically solve the different operations you would like to perform on the object, and then view of USING the class where you DON'T need to understand these details. Rather from this view, you just need to understand what information the public methods take in and how they return the results of their work.

Now, let's look at how we handle deleting a Contact. First of all, we need to search for a Contact in our list. We'll do this by name. We have an internal method which will return the index a Contact with a particular name exists in the AddressBook. If no such Contact exists, we'll return -1 to indicate that no Contact with that name is in the AddressBook. Here's the private method haveContact:

```
private int haveContact(String s) {

    for (int i=0;i<numfriends;i++)
        if (friends[i].getName().equals(s))
            return i;

    return -1;
}
```

This method is pretty straightforward: we look through the friends array and just check if the name of the ith Contact equals the formal parameter String s. If it is, we just return the corresponding index. If we complete our search and never find a Contact with that name, then we just return -1. Note the use of the getName() method. We are not allowed to say .name because name has private access in the Contact class. Rather, we must use the accessor method to get the name associated with each particular Contact object.

Now, we're ready to look at deleteContact. This will return false if no such Contact exists and no deletion was done. Otherwise, it will perform the delete and return true.

The haveContact method will return the index in the array that we want to delete. So, we call this first. If this returns -1, we just return false. Otherwise, this index value will help us.

One way to delete would be to shift everyone over 1 spot in the friends array, but if we want to be lazy, we can make the observation that the last reference pointing to a Contact should no longer do so. But, it's possible that this last reference is currently pointing to a Contact object that's still part of the address book. But, we know that we're free to use the index of the deleted item for whatever we want. So, we can kill two birds with one stone by reassigning the index in the friends array that is currently pointing to the item to delete and having it point to the last valid Contact object. Here's the code, where place represents the index in the friends array that is referencing the Contact to be deleted:

```
friends[place] = friends[numfriends-1];
```

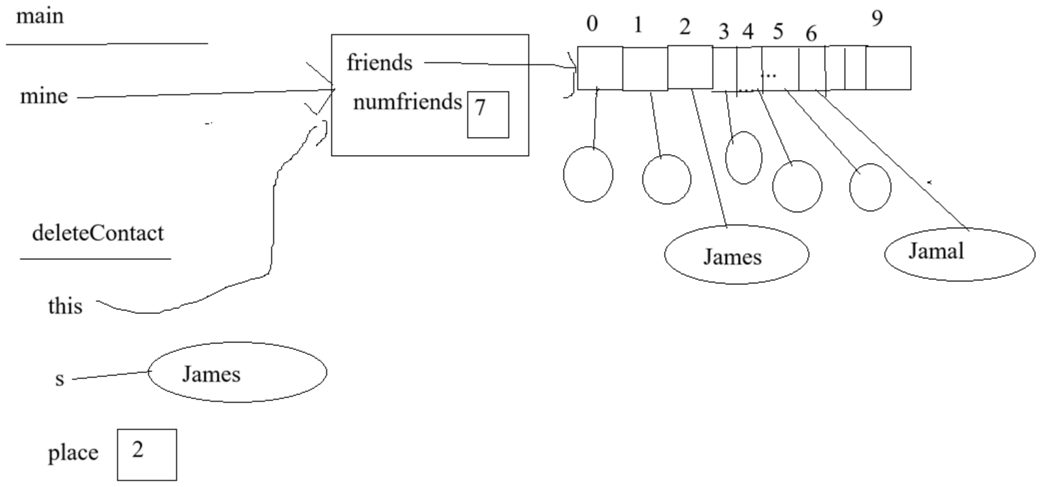
On the next page we'll show the full code and then we'll look at a single picture of what this does. After we update this reference, we also need to update the object to indicate that it stores one fewer friend, so we'll have to decrement numfriends. If we did the decrement first, then the index into the array above on the right-hand side would be different.

```
public boolean deleteContact(String s) {

    int place = haveContact(s);
    if (place < 0) return false;

    friends[place] = friends[numfriends-1];
    numfriends--;
    return true;
}
```

Consider a situation where we have 7 friends and the third friend (in index 2) is "James" who is to be deleted. Let the friend in index 6 be "Jamal". Consider the line of code

```
mine.deleteContact("James");
```

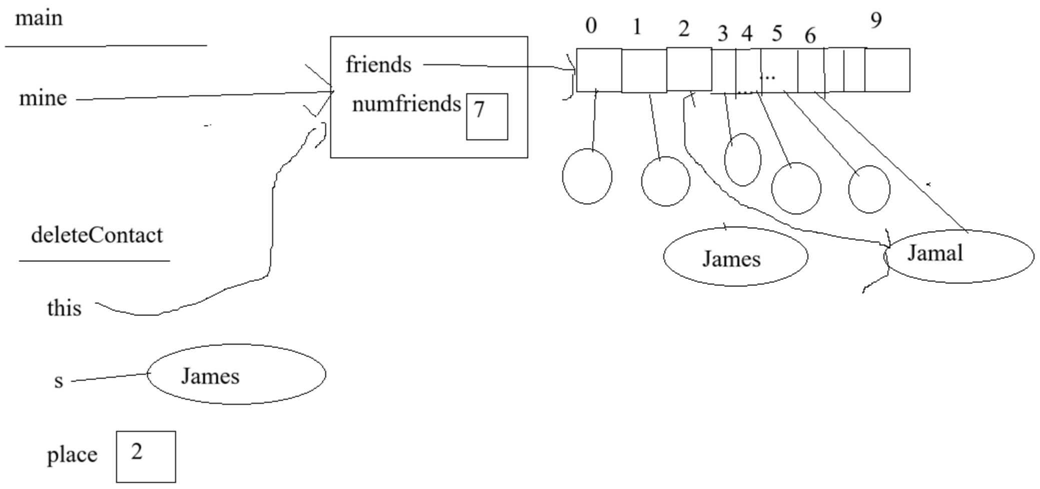
from main where mine is an AddressBook object described above. Once deleteContact starts running and haveContact completes and we complete the if statement, here is our picture (some details omitted):



Now, when we execute the line of code

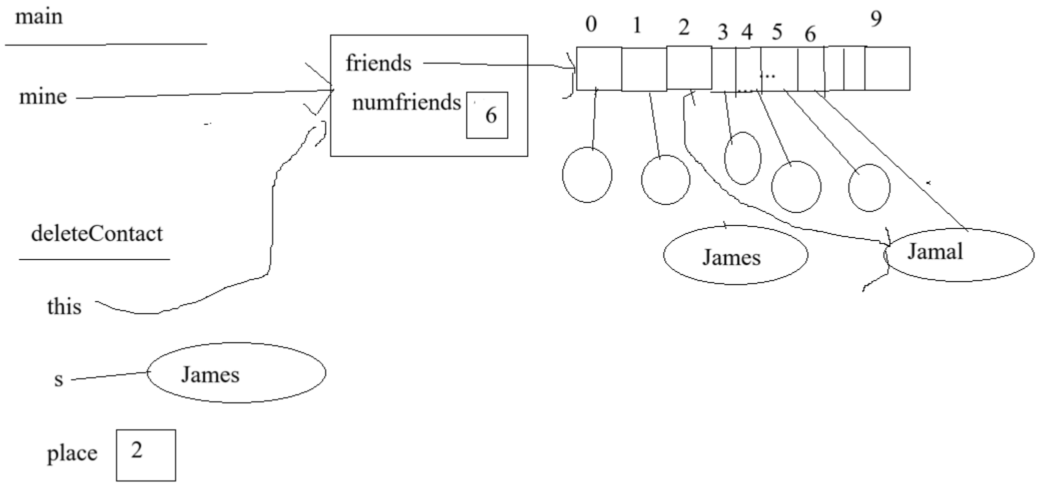
```
friends[place] = friends[numfriends-1];
```

Our new picture is:



After this statement, we have two references pointing to Jamal and none to James, so James will be garbage collected eventually. Because Jamal is now in index 2, index 6 is no longer "needed." We must update our object so that the number of friends is 6 since now we have only 6 friends:

```
numfriends--;
```



The next time we add a friend to this AddressBook, then index 6 will be reassigned to a new Contact object. Until that time index 6 will continue to reference Jamal, but index 6 won't be used at all since numfriends is 6.

One final method we provide users of the AddressBook class is the getContact method, which returns a reference to a Contact with a particular name. If no such Contact exists, then null is returned. This should be the standard return for "no object." Here we use the ternary operator to succinctly code this method, piggybacking on the private method haveContact:

```
public Contact getContact(String s) {
    int tmpIdx = haveContact(s);
    return tmpIdx>=0 ? friends[tmpIdx] : null;
}
```

Now, let's take a look at our final class, the RunAddressBook class, which just contains a static main method to help maintain a single AddressBook object. We could have just as easily made this the main method of the AddressBook class if we wanted, but this design separates out the roles of each class more clearly.

```
import java.util.*;

public class RunAddressBook {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        AddressBook blackbook = new AddressBook();
        menu();
        int choice = stdin.nextInt();

        while (choice != 6) {

            if (choice == 1) {
                if (blackbook.canAdd()) {

                    System.out.println("Enter your friend\'s name:");
                    String name = stdin.next();
                    System.out.println("Enter their age.");
                    int age = stdin.nextInt();
                    System.out.println("Enter their phone number.");
                    long number = stdin.nextLong();
                    System.out.println("Enter the birthday.");
                    int mon = stdin.nextInt();
                    int day = stdin.nextInt();

                   blackbook.addContact(new Contact(name,age,number,mon,day));
                }
                else
                    System.out.println("Sorry, can not add anyone.");
            }

            else if (choice == 2) {
                System.out.println("What is the name to delete?");
                String name = stdin.next();
                boolean res = blackbook.deleteContact(name);
                if (res)
                    System.out.println("Your contact has been deleted.");
                else
                    System.out.println("No action was taken.");

            }

            else if (choice == 3) {

                System.out.println("What is the name to look up?");
                String name = stdin.next();
                Contact c = blackbook.getContact(name);

                if (c != null)
                    System.out.println("Here is the entry you want: "+c);
                else
                    System.out.println("No entry found.);
            }

            else if (choice == 4) {
                System.out.println("#contacts= " + blackbook.numContacts());
            }

            else if (choice == 5) {
                System.out.println(blackbook);
            }
            else if (choice != 6) {
                        System.out.println("Invalid menu choice.");
            }

            menu();
            choice = stdin.nextInt();
        }

    }

    public static void menu() {
        System.out.println("1.Add a new contact to your address book.");
        System.out.println("2.Delete a contact from your address book.");
        System.out.println("3.Look up a contact by name.");
        System.out.println("4.Print out the number of contacts you have.");
        System.out.println("5.Print out information of all of your contacts.");
        System.out.println("6.Quit.");
        System.out.println("Enter your menu choice:");
    }
}
```

Notice the use of the toString method which implicitly gets called. We can retrieve the appropriate Contact in option 3 and just print it out in a regular println statement. We also use this functionality for the whole AddressBook (blackbook) in option 4. Finally notice how little the programmer using the AddressBook has to know. They literally just need to know the method names and what those methods take in. Our goal designing classes is to provide the necessary functionality for other programmers who might want to use the class, but don't want to worry themselves about the details. The pictures early in this lecture show how OOP design makes that possible.