CSE114 – Exam 3 Review

Dynamic vs Static objects

Static vs Dynamic

Static members and methods belong to the class as a whole.

Example:

ArrayTools.randomArray(size, range);

[randomArray is declared as 'static']

Dynamic:

Random r = new Random()

int i = r.nextInt(10);

r is an *instance* of Random. The call to nextInt() is from the dynamic object instance 'r'. nextInt() is declared WITHOUT the 'static' keyword so it is always called from an instance, not with the classname like ArrayTools.randomArray().

Sorting

Insertion Sort

```
public static void isort (int[] a) {

    System.out.println("\nInsertion sort:");

    for (int i = 1; i < a.length; i++) {
            int key = a[i];
            int p = i;

            //  Shift the values larger than key to the right
            while ((p > 0) && (key < a[p - 1])) {
                    a[p] = a[p - 1];
                    p--;
            }
            a[p] = key;
        }

  }
```

Selection Sort

```
public static void ssort (int[] a) {
        System.out.println("\nSelection sort:");

    // A simpler version
        for (int i = 0; i < a.length; i++) {
           // find the smallest and swap
           for (int j = i + 1; j < a.length; j++) {
                 if (a[j] < a[i]) {
                    swap(a, i, j);
                 }
           }
        }
```

Searching

Linear Search

Linear search is relatively inefficient. On average, you have to search half the elements in an array to find what you want.

```
public static boolean search1 (int item, int[] a) {
     for (int i = 0; i < a.length; i++) {
        if (item == a[i]) {
           return true;
        }
     }
     return false;
  }
}
```

Binary Search

⇨ Precondition : Array must be sorted

```
  private static int bsearch(int x, int[] a) {
     int lo = 0;
     int hi = a.length-1;
     int mid = (lo + hi) / 2;
     while (lo <= hi) {
        //          System.out.println(lo + " " + hi + " " + mid);
        if (a[mid] == x) {
           return mid;
        }
        else if (a[mid] < x) {
           lo = mid + 1;
        }
        else {
           hi = mid - 1;
        }
        mid = (lo + hi) / 2;
     }
     return -1;
  }
```

Creating Objects, Object Composition

Objects (classes) contain one or more members that may be primitive types (like int, double, etc) or other objects (String, Random, other user defined classes). This placing of objects inside objects is called Object Composition.

Classes should contain members that use 'private' as an access modifier. Access to these members should be provided (as needed) via public getter and setter methods. This allows you to control access to the internal data of an object.

Constructors are used to initialize objects when created.

Java provides a default no-arg constructor for you. However, if you define any other constructor, Java does not generate this constructor so you would have to write on yourself.

When objects are created and initialized, the memory where they reside is in the heap and an address (or reference) is returned pointing to the object. This is done when you use the 'new' operator with a class name.

Object Composition example

```
public class Account3 {

   private int balance;
   private int number;
   private Customer owner;  // Object composition here!

   public Account3 (int balance, int inumber, Customer iowner) {
      this.balance = balance;
      number = inumber;
      this.owner = iowner;
   }


….
```

Customer is another object (which is 'composed' with Account3)

```
public class Customer {

   private String name;
   private int ssn;
   private String addr;
   private int rating;

   public Customer (String iname, int issn, String iaddr, int irating) {
      name = iname;
      ssn = issn;
      addr = iaddr;
      rating = irating;
   }
…
```

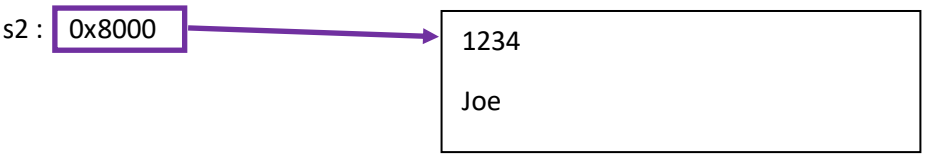
Object References, Arrays of Objects

A variable that is defined to be an object of a certain type actually is a 'reference' to the object. This is an address in memory, specifically the heap.

Student s1; // s1 will be a reference to memory holding a Student object instance. Currently it is null

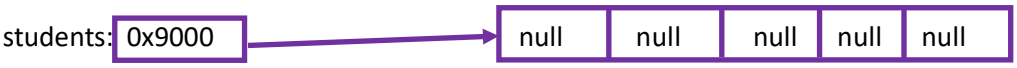
Student s2 = new Student(1234, "Joe"); // This will define s2 as a Student, initialize a new

// Student object and place a reference into s2.



You can create arrays of objects:

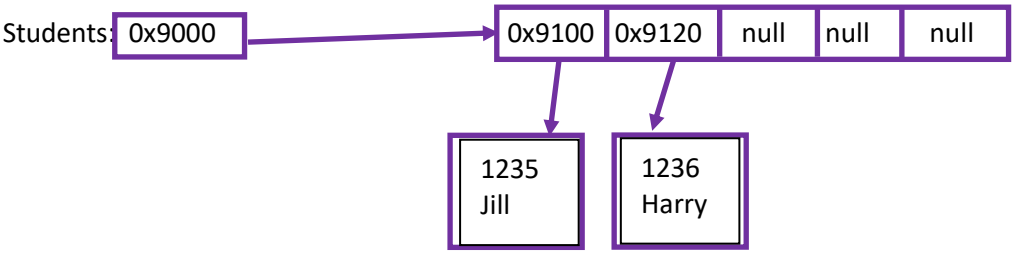
Student[] students = new Student[5];



Initially, each slot in the array will be null (point to no object). However, as you create actual Student objects, you can fill in these references

students[0] = new Student(1235, "Jill");

students[1] = new Student(1236, "Harry");



The Object Class

The Object class is the base class for all classes (either provided by the JDK or developed by users). It provides some methods including equals() and toString() which are useful for different purposes. The basic default implementations of these are not terribly useful since they do not have knowledge of the contents of an object that you define. However, you can write your own versions of these which will be used when needed.

Printing objects

For now, let's look at printing an object. If we were to print s2 from above, it will look odd:

```
System.out.println(s2);
```

Student@8000

Not terribly useful. But if you write a toString() method, you can replace that with a more useful printed representation.

The method header for toString is:

```
public String toString();
```

It must be public and return a String.

The members of Student (from the lecture notes) are id (an integer) and name (a String).

```
public String toString() {
   return "Id: " + id + ", name: " + name;
} // Note!!!!!! toString() does NOT PRINT ANYTHING. It is returning a String. That is all
```

Now when we print it:

```
System.out.println(s2);
```

Id: 1234, name: Joe

Equality testing of objects

The equals() method can be implemented in your class to properly compare two instances of your object for equality. Without this, the default implementation compares the addresses (references) to the instances. This will almost never match. Generally, we want to compare the data inside the instance.

```
public boolean equals (Student s) {
        return ((this.id == s.id) && (this.name.equals(s.name));
}
```

Make sure both fields match. For name, we can use the equals() method that is provided by the String class.

Now, we can compare actual values. Refer to students above.

```
students[2] = s2;  // Let's put Joe's record into the array also
if (students[0].equals(students[1]) {  // This will return false since they are two different records

} else {

}
if (students[2].equals(s2)) {   // This will return true since the contents of s2 are the
                    // same as the second array element. (In this case, they are the same object)
} else {

}
```

What about:

```
students[3] = new Student(s2.getId(), s2.getName());
```

Now:

```
if (students[3].equals(s2)) { // This will still be true since the contained values for id and
                              // name are the same.
} else {

}
```

Passing objects as arguments to methods

Objects (references to objects) can be passed as objects into methods. They can also be returned from methods. Note, that unlike primitive values like integers and doubles, if you change a value in an object, it will affect the copy from the calling code since it is actually the same data!

```
public void adjustId(Student s) {
   s.setId(s.getId() + 1);
}
```

This will change the value of id in the caller's Student object.

Static/Dynamic members, access control

Declaring something 'static' means the member belongs to all instances of the class (it actually belongs to the Class itself). There is only 1 copy of the member shared by all instances.

Non static members belong to an instance. If 101 instances of an object are created, there are 101 copies of the member, 1 for each instance. They live as long as the instance lives.

Local method variables exist only within the method in which they are declared. There is a different copy each time the method is invoked (called).

Variable Lifetime

Local variables: Alive while function is running only

Non-static variables: Alive as long as the instance is alive

Static variables: Alive as long as the program is alive (until main() exits)

The '.' Operator

'.' Is used to access members and methods inside an instance of an object. It is not needed inside the object's code since it is implicit that we are accessing members and methods local to the instance (class). Outside the class' code, we have to provide the name of an instance so this syntax:

```
int i = jim.getBalance();
```

Says that we want to call getBalance() inside the object instance named *jim.*

'this'

Inside of the methods (dynamic methods only) of an object instance, we can use 'this' to refer to the current instance (where we are running). The main use is to disambiguate (to distinguish) between like named variables. For instance, a constructor for student may have parameters with names that match the local members to be initialized. In this case, using *this* tells Java we are referring to the member not the parameter:

```
private int id;
private String name;

public Student(int id, String name) {
      this.id = id;
      this.name = name;
}
```

You can always name parameters different from the class members but it is standard practice to name them the same so it is clear which member they should be used to initialize or affect.

Comparing objects -> compareTo()

The compareTo() method gives you a way to compare your own classes in a reasonable way. It is similar to the equals() method but instead of returning a Boolean value, it returns an int that is:

<0 if 'this' object's value is less than the value in the object passed to the compareTo()

0 if 'this' object's value is equal to the value in the object passed to the compareTo()

>0 if 'this' object's value is greater than the value in the object passed to the compareTo()

The method header (prototype) for compareTo() is:

```
public int compareTo(Object o) {
  // Usually Object is replaced with the actual Class type where this method is implemented
}
```

So for Student, let's say we wanted a sort to order objects by id number. We can write:

```
public int compareTo(Student s) {

    if (this.name < s.name) {
       return -1;
    } else if (this.name.equals(s.name)) {
       return 0;
    } else {
      return 1;
    }
}
```