Introduction

The final exam will be given during the class time (12 to 1PM) on Friday, August 22. The exam is based on the lectures and the homework and will be very similar to the midterm in format.

The exam will be closed book, no notes, no electronic devices.

Exam coverage

The final exam will be comprehensive, meaning that it will cover material from the entire course. Much of what we have done since the midterm is apply what we learned before then, so all the material that was listed in the midterm review sheet is still appropriate for this final exam.

Study the midterm review sheet and the “questions from class” (July 23) *in addition to* this review sheet.

Files and Streams

Sources and sinks. Data read or written by a program is considered to come from a source and go to a sink. Sources and sinks can be files, memory, the console, network ports, etc. At the lowest level, every source and sink deals in bytes, but there are access methods provided by stream classes which can impose a higher level structure on the data while it is being read and written.

Streams. Getting the data from the source to the sink is the job of a stream object. Streams are subclassed and layered to provide whatever functionality is needed. There are two basic stream classes used for raw data: InputStream and OutputStream. These are abstract classes and provide only the most basic methods. The fundamental methods read() and write() are abstract in these classes and are implemented in the subclasses.

Stream subclasses. FileInputStream is an important subclass of InputStream that specifies a file as the data source. FileOutputStream is an important subclass of OutputStream that specifies a file as the data sink. These classes are concrete (not abstract) but they don't provide all the functionality that you might want at the application level.

Layered stream subclasses. Passing a stream object to the constructor of a “decorator” class provides additional functions for a stream. These classes take a stream as input to their constructor, and return a stream. The new stream has access to the methods provided in the decorator class. Examples of classes used in layering for input are BufferedInputStream, ObjectInputStream, and ZipInputStream. Examples of classes used in layering for output are BufferedOutputStream, ObjectOutputStream, and ZipOutputStream.

Readers and Writers. Since Java uses Unicode for character data, there are specialized classes provided for handling character data. Most programs should use concrete subclasses of Reader and Writer to read and write textual information. Readers and Writers can handle any character in the Unicode character set, whereas the byte streams are limited to ISO-Latin-1 8-bit bytes.

You can wrap an InputStream with a Reader and an OutputStream with a Writer as needed to get the desired formatting and desired source or sink.

End of stream on read. An expected end-of-stream condition should be detected by checking for -1 returned as the number of bytes read by the read() method, or null returned as the String read by readLine() on a BufferedReader. An unexpected end-of-stream condition will result in an EOFException, which should be caught or thrown. An exception should only occur if something unusual happens, not just that you've real all the way through to the end of the file. There will be no questions about exceptions on the exam.

File class. The File class is used to specify and manipulate pathnames. There are several overloaded constructors for the File class, providing flexibility in how directories and filenames are specified. The File() constructors create new pathname objects, not new files. There are numerous methods available to operate on a File object, including methods to check for existence and access permissions (eg, exists(), canRead(), canWrite()), create and delete files (eg, createNewFile(), createTempFile(), delete()) and to get directory information (eg, getParent(), list(), listFiles()).

Arrays

Arrays are a way to store a group of objects or primitive values and access them using an index value. An array is an object and it is created using the new operator or by providing initializers along with the declaration.

The array type is identified by giving the type of the elements it holds, followed by a pair of square brackets "[]". An array can be allocated with "new". For example

```
int[] countValues = new int[10];
```

allocates an array of integer values with 10 slots. Each of the ten slots can be used to hold an int value. The slots are accessed by using the name of the array, followed by the desired index in square brackets. The index values run from 0 to length-1. For example:

```
countValues[4] = k;
```

sets the 5th slot of the countValues array to be the same value as the current value of integer k.

Arrays can be initialized at the same time they are allocated. In this case, Java will count the number of elements for you and set the array to the correct size automatically.

```
String[] county = new String[] {“King”,“Island”,“Snohomish”,“Pierce”};
```

This statement creates a new String array object and allocates space for four slots in the array. The slots are then initialized with references to the four String literals that were given.

The number of slots available in the array is provided in the length field. For example,

```
      System.out.println(county.length);
```

would print the number 4.

It is also possible to have arrays with multiple dimensions. Sometimes we want to represent a list of groups, each of which consists of a list of other things. One way to do this is with a 2-dimensional array. In this case, the first dimension indexes the group, and the second dimension indexes the elements of the group.

The syntax for arrays with more than one dimension is similar to the 1-D case. The type of a 2-D array is the name of the element type followed two pairs of square brackets: Shape[][]. A new 2-D array is created with the new operator:

```
      Shape[][] pat = new Shape[10][5];
```

The last dimension can be left empty, in which case each entry is filled in later with an explicit new operation for each row.

```
      Shape[][] rags = new Shape[4][];
      rags[0] = new Shape[1];
      rags[1] = new Shape[2];
      rags[2] = new Shape[4];
      rags[3] = new Shape[8];
```

A 2-D array is implemented in Java as an array of arrays. There are various ways to access the elements of a 2-D array. You can access an element directly, using syntax similar to that for 1-D arrays:

```
      Shape x = pat[i][j];
```

Or, you can get the row array, then access elements of the row individually from the 1-D row array:

```
      Shape[] row = pat[i];
      Shape x = row[j];
```

ArrayLists

ArrayLists are one example of a Collection in Java. ArrayList is a class in java.util, and ArrayLists are a general-purpose container for any kind of object. ArrayLists are an ordered Collection, and so elements can be retrieved using an index to specify which one you want.

A new ArrayList object is created using an ArrayList constructor. For example,

```
      ArrayList myGroup = new ArrayList();
```

After the list is created, objects can be added to it using the add(Object o) method.

```
      myGroup.add(new Dog(17));
```

Objects can be retrieved using the get(int index) method.

```
    myGroup.get(idx);
```

The get method returns an object of type Object, so you usually need to cast the returned value to the type of the object.

```
    Dog pet = (Dog)myGroup.get(idx);
```

The number of objects currently in the list can be obtained using the size() method.

Inheritance

Extending a class allows you to build a new class by adding capabilities to an already defined class. The keyword extends is used to link the subclass to the superclass. For example, "public class PersonalFriend extends Person" starts the definition of a new class based on the existing class named Person.

If the keyword “extends” is not used in the header of the class, then the class extends Object by default. Object is the highest level superclass, and every other class is descended from Object, either directly or with other intervening subclasses. You should be able to look at a page of API documentation for a class that you are not familiar with (for example for java.util.HashMap) and be able to identify its direct superclass, the entire list of superclasses up to Object, its direct known subclasses, and the interfaces that it implements.

The subclass is said to inherit all the members of the superclass. Access to the members may be limited depending on the access keyword that was used in the member definition (private, package (default), protected, or public). We have only used private and public in this course.

Method overriding. A subclass can override a method that was defined in the superclass. In order to do this, the subclass defines a method with the same name and argument list (ie, the same signature) that returns the same value type as a method that was defined in the superclass. This allows the subclass to refine the behavior of the method to be more appropriate to the needs of the subclass. When an instance method is called at runtime, the JVM checks to see if there is a version of the method defined for the specific class of the object in use. If there is, it calls that method. If there is not, it checks to see if there is a version for the superclass. If there is, it calls that method. This continues all the way up the chain of inheritance to the "cosmic superclass", Object.

instanceof keyword. This keyword is used with a class or interface name to determine at run time if an object is of a particular type. The keyword is a logical operator giving a boolean result, and is used as follows. For an object reference o, you can test if it is an object of a particular type TestClass with the expression "if (o instanceof TestClass) {...}". If "o" is an instance of TestClass, or any of its subclasses, the conditional expression will be true. Similarly, if TestInterface is a defined interface, you can test with "if (o instanceof TestInterface) {...}" If "o" implements the interface TestInterface, the conditional expression will be true.

Abstract classes. An abstract class is one that cannot be instantiated itself, but must be extended. In general, an abstract class contains one or more abstract methods. All of these abstract methods must be implemented by a subclass that extends the abstract superclass (or the subclass itself is also abstract). The benefit of abstract classes is that they can implement methods that are common to all subclasses, while leaving other methods for implementation by the subclasses. The keyword abstract is used to identify an abstract class or method.

Static variables and methods

You should know that an application starts executing in the "main" method of the class that is specified to the java tool. You should know that the main method is always specified as "`public static void main(String[] arg)`" and that the user command line options are passed to the program in a String array parameter. In this example, the parameter is named “arg”, although it can be named anything you want to use.

Static variables and methods are defined once when a class is loaded, and they are not repeated when an object is created. This can be useful for defining utility methods that are not associated with any one object, and also for creating a static variable that applies to all members of the class, rather than to each individual object of the class. Static methods and variables are accessed by preceding the member name by the name of the class, rather than the name of the object reference. Thus, to access a search method in the Arrays class, you could type Arrays.binarySearch(a,key), and to access the maximum Integer value possible you could type Integer.MAX_VALUE.