Introduction

The 1-hour midterm exam will be given during the class time (12 noon to 1PM) on Friday, July 26.

The exam will not be in the regular classroom, it will be in Guggenheim 224, the building just to the north of the EE1 building where we usually meet.

The exam will be closed book, no notes, no calculators. Review the homework questions and the class notes, and read this document carefully, and you will do fine on the exam.

Cheating is stupid. Do your own work! You'll learn lots and hopefully enjoy the process. Study the class materials and ask questions if anything is unclear.

The exam is based on the lectures and the homework. The questions will be similar to the questions on the homework with several small programming questions too.

Objects

Objects have state and behavior. State is maintained in instance variables which live as long as the object does. Behavior is implemented in methods, which can be called by other objects to request that something be done.

Objects are instances of classes. Classes are initially defined in a source file that ends with .java. The compiler reads the source file and converts it into a binary file for use in the program. The binary class file is the same name as the source file, except that it ends with .class.

Classes

A class is the fundamental unit of a Java program. A class file is the template or blueprint from which objects are built.

We use the "new" operator to create (or instantiate) a new object from a class definition. The new operator allocates memory for the new object, then calls a "constructor" in your code to fill in all the details of the state of the object and do any other initialization that is required.

The layout of a Java class definition source file is defined by the blocks of code that it contains. Curly braces define individual blocks of code. Blocks can be nested. The order of some of the elements below can be changed, but this is a general guideline.

In general, the outermost block is the class block. This is something like "public class MyClass { .. body of the class …}. Everything else that follows is part of the "body of the class" and is located between the outermost curly braces.

The first inside block is usually a constructor. It might be something like "public MyClass() { … body of the constructor …}. A constructor is just like a method, except that it is called only when a new object is being created, and it doesn't explicitly return a value. It is the interior decorator that fills in all the details before handing over the newly created object to whatever piece of code used the "new" operator to request the new object.

There may be zero, one, or more constructors. Each one is a block. The order of the constructors doesn't matter.

After the constructors there are usually some methods. They will be something like "public int getMealsize() { … body of method …}.

I usually put the public methods (which other classes might want to call) ahead of the private methods (which are just for use in this class). It doesn't matter to the compiler, but another programmer is likely to be more interested in the public methods than the private methods.

After all the methods and just before the end of the class, I put the instance variables. Some people put them up at the front of the class, before the constructors. Either way works. I put them at the end because I think they are details that the casual reader does not need to see right away. The instance variable declarations are not inside any constructor or method.

Comments in the code

Java has an excellent tool named javadoc for extracting comments from the code and building well organized web pages from it. A typical set of comments for a method will include a description of the method, an @param entry for each parameter variable, and an @return entry if there is a value returned from the method. I expect that you will be able to write a brief set of comments for a given method, and be able to write a method, given a good set of javadoc comments. An example set of comments:

```
/**
 * Decide if the other Shape intersects any of the Shapes that
 * this object is displaying.  This method runs through all of the
 * Shapes that are being used to represent this object, and uses
 * the intersects(other) method to decide if there is an
 * intersection.
 * @param other the other Shape that we might intersect with
 * @return true if the given Shape intersects one of our Shapes,
 * else false
 */
 public boolean intersects(Shape other) {…
 }
```

Variables

Values in the program are kept in named variables. There are two types of values: primitive types and reference types. Primitive types refer to simple values like integers and floating point numbers. Primitive types have no behavior (no methods) associated with them.

The type "int" is used to store numeric values with no fractional part. The values can be negative or positive, and range from a very large negative value to a very large positive value. There is a constant defined for each of these values: Integer.MIN_VALUE and Integer.MAX_VALUE. There are other integer types besides "int" but we have not discussed them in this class.

The type "double" is used to store numeric values that may have a fractional part or that are too big to keep in an integer format. There are other floating point formats besides "double" but we have not discussed them in this class.

The fractional part of a floating point number holds approximately 16 decimal digits. This is plenty for most numeric calculations, but it is only an approximation to a real numeric value which may contain many more digits or even an infinite number of digits.

The fractional part of a floating point number is truncated when it is stored in memory. Therefore, floating point numbers are an approximation and should not be relied on to act exactly like real numbers do in all cases. For example, the following code prints "`y does not equal 1.0.`".

```
double y = 1/3.+1/6.+1/3.+1/6.;
if (y == 1.0) {
      System.out.println("y equals 1.0.");
} else {
      System.out.println("y does not equal 1.0.");
}
```

You should never use the "==" operator to compare floating point numbers. There is usually some roundoff error way down at the end of the fractional part and the values will not be exactly equal and so the comparison will fail.

The type "boolean" is used to store values that are either "true" or "false". boolean variables can be given a value using the true and false literals, or they can be assigned a value as the result of an expression like this.

```
boolean isReady = (a==1) || (z==0);
```

The second group of Java types is the reference type. The values of a reference type are references to objects. An object is an instance of a class definition, and a reference to that object is a way to remember where it is and to ask it to do things. More discussion of objects and classes below.

Kinds of variables

We have talked about three major kinds of variables in this course.

Parameter variables are the means by which the caller provides information to the constructor or method it is calling. In a method, you declare the parameters you must have by listing them in the parameter list for the method, which is the part in parentheses right after the name of the method. For example, in "public Dog(String theName)" we are telling the compiler that anybody that wants to use this constructor must supply a String value to us, and that we will refer to that String using the variable name "theName".

Local variables are the way that a constructor or method creates little scratchpad areas to use as it does whatever its task is. Local variables are declared within the body of the constructor or method. Local variables are not initialized automatically, so you must make sure that you do the initialization explicitly. Local variables are thrown away when the body of the method is finished executing, so there is no carryover of value from one execution to the next in a local variable.

Instance variables are the way an object keeps track of its state. *Each* object gets its own copy of the instance variables for its class. Instance variables are declared outside the body of any constructor or method (but within the body of the class). Instance variables retain their values as long as the object exists.

Statements and Expressions

The body of a constructor or method is built up from individual statements. Each statement ends with a semi-colon ";". Statements can be grouped into blocks using curly braces "{" and "}".

An expression describes how to compute a particular value. Expressions can be formed using literal values (eg, 4, false, "Tweet"), the current value of a variable (eg, cornerX, isReady, name), the result of a method call (eg, myShape.getX(), myAnimal.isReady(), myAnimal.getName()), and the result of the new operator (eg, new Dog(17)).

The arithmetic operators for add, subtract, multiply, divide, remainder are described in the lecture of 1-July. They all act more or less as you would expect. Pay particular attention to integer division – since the result is an integer, an expression like (1/2) gives the result 0. This can be surprising if you don't expect it.

Methods

The behavior of an object is defined by the methods in the class. All the objects created from a particular class template share the same methods. The actual behavior of the objects may be somewhat different because they each have their own state variables, but the methods are the same for all objects of the same class.

You pass information to a method in the parameter list. The method defines the values that it needs in the method header, and then the caller must supply the correct number and type of values in the actual call to the method. You should be able to read and write method headers (eg, “public void eat(double amount)” and understand where the return type is defined, where the name of the method is given, and the meaning of the parameter list in parentheses.

The variable names that are supplied in the parameter list are used in the body of the method to refer to the values that were supplied by the caller. The name that is used does not have to match the name that the caller is using. In fact, the caller can supply literal values, or values that are the result of an expression or another method call.

Methods can return a value to the caller if appropriate. If there is a return value, the type of value returned is specified right before the name of the method in the method header.

Constructors

The Java system calls a constructor for class XYZ when another part of the program wants to create a new object of class XYZ. Java allocates a little bit of memory for the new object (space for the instance variables) then calls the constructor to fill in the details and do whatever initialization is needed. Constructors can be thought of as a special type of method that gets called only when a new object is being created. There is no return type because the only purpose of the constructor is to initialize the new object.

The name of a constructor is always the same as the name of the class. The parameter list for the constructor can be whatever is you need. It is common to define one constructor with no parameters at all that uses default values for every instance variable. Additional constructors (same name, different parameter lists) can be defined that take various values that the creator might want to have specific control over. For example, there are three constructors for java.util.ArrayList. One of them creates an empty list of default size. One of them creates a list that contains everything that some other existing Collection contained. The third one creates an empty list of a specified size.

Decisions

The “if” statement is the primary means for checking a condition and changing the program flow based on whether or not it is true. The format of an “if” statement is:

```
if (condition) {
      this block executed if condition is true
} else {
      this block executed if condition is false
}
```

The else block can be left out if appropriate. This statement can be used without the curly braces if the block to be executed is only one statement, but it is generally a better idea to use the braces all the time so that your code is consistent and can be more easily updated. “If” statements can be chained together using if (…) {} else if (…) { …}

All of the standard numeric comparison operators are available: `>, >=, <, <=, ==, !=.` These symbols represent: greater than, greater than or equal to, less than, less than or equal to, equal, not equal. Logical expressions can be combined by using the “and” (`&&`), “or” (| |), and “not” (`!`) operators to make larger expressions.

The conditional operator (a?b:c) is an easy way to check the truth value of “a”, then use value “b” if “a” was true, and value “c” if “a” was false.

Never use the equals operator “==” to compare two floating point values. Two values that are very, very close may well differ in one very insignificant bit and therefore fail the comparison. Use “greater than or equal”, or “difference less than epsilon” or some similar construct to make decisions about floating point values.

Looping

The “for” and “while” statements are the primary means for executing loops in our programs. You can generally consider loops to be either counting a certain number of times (usually a “for” loop) or looping until a certain condition is true or false (“while” loop). However, it is easy to recode a loop that is written as a “for” loop to be a “while” loop, and vice versa, so don’t get too concerned about the “correct” way to code a loop. Use whichever construct makes the most sense to you, given the desired actions of the code.

The “for” loop is written as one statement with a body that is executed over and over. In the parentheses of the for loop there is space for three statements: the initialization, the check for termination, and the end of loop action. Make sure you understand the purpose and operation of each statement.

The “while” loop is written as one statement with a body that is executed over and over. In the parentheses of the while loop is the condition that is checked before every loop. If the condition is true, then the loop body is executed once, and the condition is checked again.

If the condition if false, then the "while" statement is completed and the next statement after the body of the while loop is executed.

Depending on the values of the control variables, it is entirely possible that a for loop or while loop may not go through its loop code at all, but rather just skip over the whole thing.

It is possible to leave a loop statement early if conditions warrant using the break statement. This might be useful if you want to search through a list of objects, and stop searching if you find something in particular. The break statement takes you completely out of the "for" or "while" loop.

It is possible to skip to the end of a loop statement if conditions warrant using the continue statement. This might be useful if you want to search through every object in a list, but only perform work on a few particular objects. The continue statement jumps to the end of the loop and gets ready to start another pass, just as though you had executed all the code in the body of the loop and gotten to the end.

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.

Interfaces

An interface defines a set of methods that a class can define and then tell the world that it implements the interface. This tells every other class that, no matter what else this class does, it at least implements the methods defined in the interface.

Once a class has said that it implements an interface, then objects of this class can be referred to as though they were of the type named by the interface, rather than whatever their class name is. See the Interfaces lecture of July 15, slide 20, for an example of this. This is a very nice feature, because it means that the caller can treat all objects that implement an interface the same way, regardless of their actual type. This simplifies the code.

The keyword `extends` can be used in an interface definition to refer to another interface. By extending an interface, the definition is saying that an implementing class will guarantee to implement both the original interface methods, plus any methods defined in the extension.

Tools

You use an editor to write the java source file. This file is named the same as the class it defines, and ends with ".java". The compiler reads the source file and compiles it, producing the binary class definition file. This file is also named the same as the class it defines, but it ends with ".class". The java virtual machine (JVM) reads the class files and executes the code they contain. Execution starts with a static method named "void main(String[] args)" in the class that you specify when you request that the JVM run your program.

We are using BlueJ as an editor, and to control the compiler and JVM. Many other editors are used in Java program development, and most java application programs are run directly by starting the JVM and specifying a class name to start in, rather than running with BlueJ managing the interaction.