Modularization

The next coding assignment, Adventure, will be significantly more complex than TicTacToe and JSON. However, just because the functionality provided by the program is more complex, doesn't mean that the code has to be more confusing / more difficult to digest and understand.

The key to writing *simple* code that performs *complex* tasks is modularization (i.e. splitting up the functionality in a logical and intuitive way). One reason code can become difficult to understand is that there are too many steps / details to keep in your head at the same time, and you lose track of the big picture. Modules alleviate this problem by hiding details behind an interface.

When you use a module, you don't have to worry about the details happening behind the scenes; you should only have to worry about the result (e.g. the return value, or the change in an object's state). Conversely, if a module requires you to know what's happening behind the scenes in order to use it, then it hasn't been designed well. (For example, if function1 calls function2 in your code, someone else reading function1 shouldn't have to stop and read function2 in order to understand function1).

Functions are modules

The most fundamental level of modularization is functions. Functions should make sense as a single "unit" of work (i.e. can't be split up into smaller units of work that make sense on their own). Each method should perform one distinct task. If you find yourself writing very long functions, then it can probably be split up. Long functions require you to keep more details in your head, which can lead to more bugs. This problem is exacerbated if the function no longer fits on a screen and you have to scroll to read it. Short functions are also easier to test thoroughly, and when a test fails, you'll have more specific information about what exactly went wrong.

Object decomposition

Functions are the smallest unit / module in our code. Objects/classes are another level of modularization that is one level higher. They are a way to bundle functions together into larger coherent units. (For example, it might make sense to group methods in an object if they both operate on data stored in that object's member variables.)

Big idea: The purpose of object decomposition is to group related things together and separate unrelated things.

Choice of classes to create

When creating good object oriented code, it is crucial to think carefully about which classes you need. Classes should be coherent units and have clear purposes, and avoid covering functionality that isn't their 'job'. Think about what one class is responsible for doing and try to make sure it is only executing that functionality.

Separating input/output handling from game logic

One important aspect of modularization is to prevent input/output handling from being too entangled with the actual game logic. In a future assignment, you will be extending Adventure by allowing a user to play from a website instead of the command line. You should be able to hook up your game engine to a different type of input/output without any significant changes. In order to achieve this, the game engine class must provide an interface that can be used by multiple clients (both the command line and the web server). In other words, the game engine class shouldn't be responsible for too many things; it'd probably be a good idea to *create a separate module* for parsing user input.

In general, modularization makes our programs more flexible, because we can reuse our modules in different contexts.

Parallel Arrays

Having parallel arrays to track information is generally bad practice and can easily lead to inconsistency and bugs. If you find yourself creating parallel arrays, it's usually a sign that you may want to group that information in a class. For example, instead of having 3 arrays, xCoor, yCoor, and zCoor to keep track of a collection of points, we can define a Point class that has 3 member fields: xCoor, yCoor, and zCoor. This is a lot less prone to error, since we don't have to concurrently modify 3 lists when we want to insert, delete, or reorder points.

Placement of Methods

How to determine which class a method belongs in?

Figuring out which class variables/methods “belong” to can be tricky; it’s an intuitive skill that develops with practice. For example, consider the following code:

```
public class Zoo {
      List<Animal> animals;
}

public class Animal {
      private int age;

      int getAge() {
            return age;
      }

      int getOldestAnimalAge(List<Animal> animals) {
            // implementation here
      }
}
```

In this case, getOldestAnimalAge is in the class Animal, which may seem appropriate. After all, it returns the age of an animal, and takes a list of Animals as a parameter. But we want to consider more than that. Getting the oldest animal’s age is an operation that we perform on a Zoo and on the Zoo’s data; it does not make sense as an operation performed on a single Animal. We could restructure the code to look like this:

```
public class Zoo {
      List<Animal> animals;

      int getOldestAnimalAge() {
            // implementation here
      }

}

public class Animal {
      private int age;

      int getAge() {
            return age;
      }

}
```

Note that this also reduces the number of arguments needed for the function, because the data that the function operates on is *already stored* in the same class as the function. This makes the code simpler and also better encapsulated (because the Zoo class doesn't need to pass data around to other classes).

Member variables stored by each class

Redundant member variables / minimize the amount of state in your objects

In general, it’s best to avoid redundant member variables that contain multiple copies of the same data, or data that can quickly be derived from other data. For example, in the class below:

```
class GameEngine {
    int numberOfPlayers;
    List<PlayerStrategy> playersList;

    // member functions
}
```

The member variable numberOfPlayers is redundant when you also have a list of players. This is not *safe from bugs*, because you have to remember to update all copies of the redundant data, and you might forget to update one of them. For example, if you want your GameEngine to allow a player to join in the middle of a tournament, perhaps you would add their strategy to the list, but forget to increment numberOfPlayers, which would almost certainly cause a bug.

A better solution that maintains the same level of readability is to make a very simple member function:

```
class GameEngine {
    List<PlayerStrategy> playersList;

    int getNumPlayers() {
        return playersList.size();
    }
}
```

Encapsulation

Encapsulation is the concept of bundling data and functionality into individual units, and hiding the internal state of an object -- essentially creating a firewall.

Public vs. Private

We use access modifiers to hide unnecessary details from the client of a class. Making member functions and variables private strengthens the firewall. Only methods which are needed by clients should be visible as part of the public interface of the class. This makes the module easier to use (since it's easier to find the functions which are actually relevant).

Hiding helper methods and member variables also ensures that clients don't rely on the internal representation / implementation of the class. This allows the representation / implementation to be changed without breaking any clients' programs, as long as the changes still satisfy the interface's specification. For example, a class that represents a complex number could change its internal representation from Cartesian to polar; this change would only be safe if the member variables were private.

Don't need getters and setters for everything

In order to make the firewall as effective as possible, we should minimize the amount of entry points from which clients can access/change member variables. In other words, we don't need a getter and a setter for EVERY private variable. We only need a setter if it makes sense for clients to have write access to the variable; similarly, we only need a getter if it makes sense for clients to have read access to the variable.

Mutable private variables

Imagine that class A has a private member variables. That variable should generally only be modified by member functions which are *also* in class A (that's what encapsulation means). Code from a different class (say, class B) shouldn't be directly handling class A's member variables. (This is bad because if class B handles class A's member variables, then class B has to deal with the details of class A's internal representation, which destroys the firewall between modules.) Instead, class B should only modify class A by calling class A's public member functions.

Here's a slightly tricky example:

```
class Tournament {
    private List<Integer> scores;

    public List<Integer> getScores() {
        return scores;
    }
}

class Game {
    public void updateScores(tournament) {
        for (int i = 0; i < getNumPlayers(); ++i) {
            tournament.getScores()[i] =
                calculatePlayerScore(players[i]);
        }
    }
}
```

Here, Game is directly modifying the Tournament's scores member variable. This is possible because the getScores() method returns a reference to a ***mutable*** private member variable. Doing that is essentially the same as making your member variable public, which is really bad in terms of encapsulation.

Instead, the Tournament could call the Game's calculatePlayerScore method, and then take care of updating the overall tournament scores.

In general, mutation can be dangerous, especially when it's not clear that mutation is occuring. It can make code more difficult to understand (because it's hard to find how different objects are communicating with each other), and it can make code more prone to bugs (because your variables might get unexpectedly mutated without you knowing about it). Usually, it's better practice to have a function communicate its results via its return value. A void function mutating an object might not be clear, even with good naming practices, and can make debugging (as well as testing) hard.

Testing Objects

When we design objects, we want a way to execute a small step at a time (e.g. a single turn in a game). If the interface of the object only allows you to run an entire game all at once, it’s basically impossible to determine if everything worked properly just based on the result of the game.

If an instance method modifies the state / member variables of an object, you not only need to check the return value of the method; you also need to make sure that the state of the object has updated properly!