Sunday, December 30, 2018 2:04 PM

Encapsulation

The modern world depends on a concept called "abstraction" that separates how to use something from the details of how it's implemented. You can press the brake pedal of a car and stop effectively without knowing anything about disc brakes, calipers, hydraulics, or frictional coefficients – thank goodness!
In object oriented programming this separation of how to use code from its implementation details becomes important in facilitating relationships between code entities. Imagine you are writing an object that a coworker is going to build off of in their code, it would take forever if everyone had to understand every single line in the code base just to be able to use eachother's work! Instead you want to clearly separate the code in your object that other code will use from the intricate details of how you maintain your object.

The way that you define this separation is by "hiding" implementation details from view with something called "encapsulation". Encapsulation, or information hiding, was perhaps the first major step forward when Object Oriented programming emerged in the 1980s, allowing one team to use the products of another team without having to understand how it worked internally.
Encapsulation also allows you to strictly control how a user interacts with your object. By hiding all the inner workings of your object from the user you can guarantee that your object stays in a good state.

For example, you can hide your fields from the user so that you are the only one who can set their values. This can prevent your user from accidentally setting field values that don't make sense.

Just imagine if you didn't have a brake pedal and everyone driving was expected to know how to apply pressure to the disk breaks, chances are a lot of people would ruin their cars! By hiding the inner workings of the car with the casing and providing a clear way to interact with the brakes (the pedal) you help keep the car in working order.

Public vs Private

If "encapsulation" is the act of "hiding" things from your user, how do you set the visibility of code elements?
This is where the public and private keywords come into play.
You're probably familiar with the "public" keyword as we have been using it in front of all our methods and constructors up until now. Instead of making everything "public" a well encapsulated object only exposes those things that a client needs to use the object effectively, making everything else "private". This is how you prevent a user from accessing things they are not supposed to use.
Obviously, some things must be public or no one could use the object! But now by using "private" and "public" intentionally you can decided exactly how external code should interact with your object.

Private Fields

```
public class MyClass() {
   private String field1;
   private int field2;
   private double field3;
   ...
```

As they store the foundational information about your object fields should generally all be made private. This way you can use them like normal variables within your class, but your client code cannot change their values directly. This can protect the inner state of your object from being corrupted.
If you want your client code to know the current value of a field, you'll want to write separate public methods that limit the way a user interacts with your fields. That way you can protect your object in several ways:

- Restrict the value of fields to a range of valid values (maximum speed, minimum price, only certain values, etc.)
- Make sure that values of different fields remain consistent.
- Prevent clients from making assumptions about field values that you may not have intended.

Let's say we are writing a class that represents the current state of a Student within a school.

```
public class Student {
   private String name;
   private double gpa;
   private int daysAbsent;
   private boolean canGraduate;
}
```

These pieces of information are all either very private, or have very specific limitations on what is considered "valid" values. For example, in some schools a GPA can only be between 0.0 and 4.0, but if you leave that field private client code

could set it to a number that doesn't make sense.

Private Methods

private void myMethod() {

When a method is declared as "private" it means you can only call that method from within the class in which it is defined.

Usually, methods are public unless they are an internal "helper" method that is used by other public methods. These private methods can help you remove redundancy and clean up your own work. They are not for external users to have access to behavior, but rather a way for you to internally structure your own code. Let's say we are writing methods for our Student class, and we find that we keep repeating the check to see if they are eligible to graduate. This is a check you might have to do often, but is such an important state you might not want external code to be able to perform that update. If you make a private method you can call if within your class to eliminate redundancy, but you protect your Student state from being updated by a client class.

```
private boolean updateGrad(double gpaUpdate, int attendanceUpdate) {
   gpa = (gpa  + gpaUpdate) / 2;
   daysAbsent += attendanceUpdate;
   canGraduate = (gpa >= 2.0) && daysAbsent < 10;
}
```

Private Constructors

private MyClass() {

Rarely will you want to make a constructor private, as the constructor is what enables your user to even create an instance of your object. However, if you are overloading constructors and you are using a constructor that sets each individual field to eliminate redundancy you might want to make it private and only expose the constructors that take the correct parameters for your user.

```
private String name;
private double gpa;
private int daysAbsent;
private boolean canGraduate;

private Student(String name, double gpa, int daysAbsent, boolean canGraduate) {
   this.name = name;
   this.gpa = gpa;
   this.daysAbsent = daysAbsent;
   this.canGraduate = canGraduate;
}
```

```
/* Create a new student at the beginning of the school year */
public Student(String name) {
   this(name, 4.0, 0, true);
}

/* Create a new student joining later in the school year */
public Student(String name, int daysAbsent) {
   this(name, 4.0, daysAbsent, true);
}
```

Accessing Private Elements

When you make a code element "private" you can use it within the class in which it is defined, but not in any client code or other object definitions. However, you can access private elements of other objects of the same type.

```
/* this method works even though the other Student's fields are marked as private
*/
public boolean equal(Student other) {
   return this.name.equals(other.name) && this.gpa == other.gpa &&
this.daysAbsent == other.daysAbsent;
}
```

Accessors and Mutators

To protect your object's data most often you will make fields private. However, you typically still want to allow your user a way to interact with the state of your fields.

Accessors

Accessors are a type of method that exposes the current value of your field to external code. You can think of these as a "read only" method for your field's values. Often they are referred to as "getters" because typically their names are getFieldName() like so:

```
public double getGpa() {
   return gpa;
}
```

It's common for you to have an accessor method for each of your fields. They typically do not take in any parameters and simply return the current value of the field they are accessing.

Mutators

Mutators

Mutators are a type of method that allows external code to change the value of a field. This is a lot of responsibility, so often you will want to add logic to these methods to maintain a good state for your fields.

```
/* allows the user to change a student's name, accepts all types of capitalization.
Names cannot contain spaces */
public void setName(String name) {
   if (name.indexOf(" ") != -1) {
      throw new IllegalArgumentException("names cannot contain spaces");
   }
   this.name = name.toLowerCase();
}
```

Mutator methods typically take in a parameter and change the value of a field after doing some checks to see if it is an appropriate value.