Encapsulation: The Second Principle

What is Encapsulation?

Encapsulation is the act of enclosing something, as if it were in a capsule. It is hiding the details of something so that other code can't see or manipulate them. More to the point, it is hiding the sources of change. Consider the following code.

Python C#

```
class Account:

    def __init__(self):
        self.balance = 0

    def deposit(self, amount):
        self.balance = self.balance + amount
```

Python C#

```
account = Account()
account.balance = 50
account.deposit(100)
```

What would happen if we decided to change the balance attribute in the Account class to a list of transactions? On one hand, we'd be able to keep track of individual deposits. On the other hand, we'd break a different part of the program.

Python C#

```
class Account:

    def __init__(self):
        self.transactions = []    # if we change this...

    def deposit(self, amount):
        self.transactions.append(amount)
```

Python C#

```
account = Account()
account.balance = 50          # ...this doesn't work anymore!
```

```
account.deposit(100)
```

In this case, the details of the Account class are not well encapsulated. We need a way of hiding the class attributes so that other code does not attempt to change them directly. If we can, it will allow us to minimize the interdependencies between different parts of our code and protect them from breaking changes.

Using Access Modifiers

Some programming languages, like C#, use access modifiers, or special keywords, to specify which attributes and methods are public or private. Public class members are accessible from anywhere in the program. Private class members are only accessible by methods in the class that contains them. Other languages, like Python, use coding conventions to indicate how a class member should be treated.

Python C#

```
class Account:

    def __init__(self):
        self._transactions = [] # the "_" prefix means treat this as private

    def deposit(self, amount):
        self._transactions.append(amount)
```

Sometimes it's difficult to decide what class members should be public or private. A good rule of thumb is to restrict access to class members as much as possible. Attributes should be private. Methods should only be public when there is a specific need. Note that private methods are often called "helper" methods. They are used to define repeated operations or decompose larger ones into smaller tasks.

On Getters and Setters

Getters and setters are simple methods that allow outside code to access and change attributes directly. They are called getters and setters because they are usually written get<attribute name> and set<attribute name>. They are so common that many well known IDE's will automatically generate them for you. Consider the following code.

Python C#

```
class Account:

    def __init__(self):
        self._transactions = []

    def get_transactions():
        return self._transactions
```

```
    def set_transactions(transactions):
        self._transactions = transactions
```

In this case, the getter and setter make the transactions attribute effectively public. Methods like these have been the subject of debate since at least 2003 when Allen Holub published an article called, Why Getter and Setter Methods Are Evil.

While we encourage you to develop your own opinion we think Mr. Holub's advice is sound. When you're programming with classes, focus on *what* the class must do rather than *how* it will do it and many of the getters and setters in your code will naturally disappear. You simply won't have to worry about it.

Closing Thoughts

Encapsulation is the second principle of programming with classes. Remember it is about hiding information. Careful use of access modifiers, through convention or keywords, will help protect you and your coworkers from making breaking changes to the code.

The benefits don't stop there though. Practice encapsulation diligently and your abstractions will become more refined, your objects more purposeful, and your classes more understandable. Continue to work at it, and over time, your programs will be vastly more flexible and easy to change.