CS61B Lecture 7

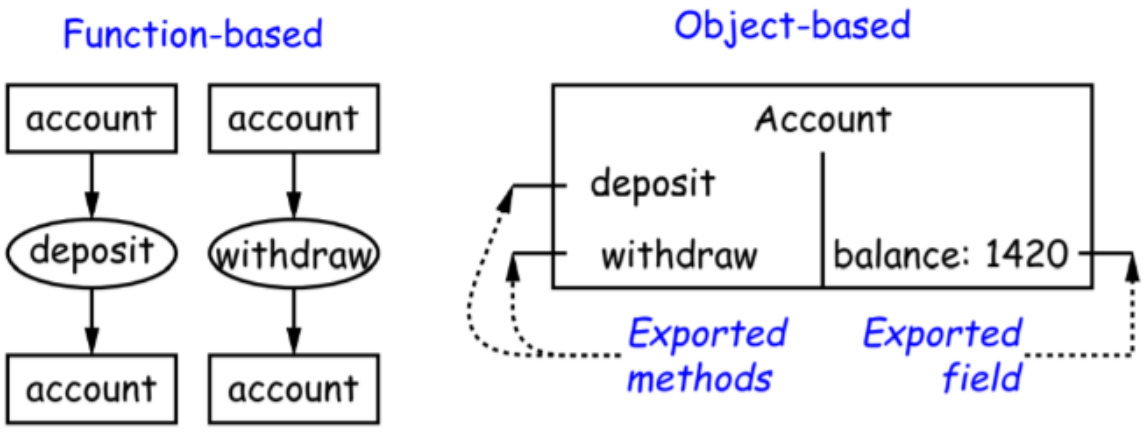
Wednesday, February 5, 2020

Object-oriented programming

Back in the 1950s, computer programs got larger and larger, and became more difficult to read, handle and update. Computer scientists got into discussions as how to best handle this. One rather unhelpful suggestion was to write smaller programs.

Object-oriented programming was developed in the late 1950s and early 1960s as a way to make it easier to manage large programs, and have programs more closely simulate the real-life objects they represent.

Object-based programs are organized around the types of objects that are used to represent data; methods are grouped by type of object. Function-based programs are organized primarily around the functions (methods, etc.) that do things. Data structures (objects) are considered separate.



To read more about object-oriented programming, see my CS61A Notes.

In general, we prefer a purely procedural interface, where the functions (methods) do everything —no outside access to the internal representation (i.e., instance variables). That way, implementor of a class and its methods has complete control over behavior of instances.

In Java, the preferred way to write the "operations of a type" is as instance methods.

A comparison against Python

Here is the `Account` class from CS61A (in the notes above!):

```
class Account:
    balance = 0
    def __init__ (self, balance0):
        self.balance = balance0
    def deposit(self, amount):
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
```

```
        if self.balance < amount:
            raise ValueError("Insufficient funds")
        else:
            self.balance -= amount
            return self.balance

myAccount = Account(1000)
print(myAccount.balance)
myAccount.deposit(100)
myAccount.withdraw(500)
```

And here is Java:

```
public class Account {
    public int balance;

    public Account(int balance0) {
        this.balance = balance0;
    }

    public int deposit(int amount) {
        balance += amount; return balance;
    }

    public int withdraw(int amount) {
        if (balance < amount) {
            throw new IllegalStateException("Insufficient funds");
        } else {
            balance -= amount;
        }
        return balance;
    }

}

Account myAccount = new Account(1000);
print(myAccount.balance)
myAccount.deposit(100);
myAccount.withdraw(500);
```

> **What happens if we change the declaration of `balance` to static?**
> That means `balance` becomes a variable that is shared among all instances of `Account`!
> Static variables are what we know in Python as class variables.

Getter Methods

Slight problem with Java version of Account: anyone can assign to the balance field. This reduces the control that the implementer of Account has over possible values of the balance. You're not supposed to be able to change your own balance like that!

Solution: allow public access only through methods:

```
public class Account {
    private int balance;
    ...
    public int balance() {
        return balance;
    }
...
}
```

Now `Account.balance = 1000000` is an error outside the `Account` class. (In 61B, we use the convention of putting `_` at the start of private instance variables to distinguish them from local variables and non-private variables. Could actually use balance for both the method and the variable, but please don't.)

Instance and Static Methods

An instance method can be taught of as a fancy kind of static method, featuring hidden arguments:

```
int deposit(int amount) {
    _balance += amount;
    _funds += amount;
    return balance;
}
```

can be thought of as writing this piece of code:

```
static int deposit (final Account this, int amount) {
    this._balance += amount;
    this._funds += amount;
    return this._balance;
}
```

Although we should note that the above code is not legal Java, as you are not allowed to declare `this` as an instance variable.

Calling Instance and Static Methods

Calling such an instance method:

```
myAccount.deposit(100);
```

is equivalent to this call on our fictional static method:

```
Account.deposit(myAccount,100); // which also is not valid Java
```

Inside a real instance method, you can leave off the leading `this.` on field access as long as your call is not ambiguous, unlike Python, which will throw up an error if you try to call an instance method without `self.`

Real static methods don't have the invisible `this` parameter, which means we can't refer directly to instance variables inside of them:

```
public static int badBalance(Account A) {
    int x = A._balance; // This is OK
    return _balance; // This is nonsense
}
```

The call to `_balance` is equivalent to `this._balance`, but this static method has no sense of what `this` is. Java will throw up some error that will say "trying to access instance variable from a static context".

Constructors

To completely control the objects of some class, you must be able to set their initial contents. A **constructor** is a special kind of instance method that is called by the **new** operator right after it creates a new object:

```
L = new IntList(1, null);

tmp = pointer to [0,\];
tmp.IntList(1, null);
L = tmp;
```

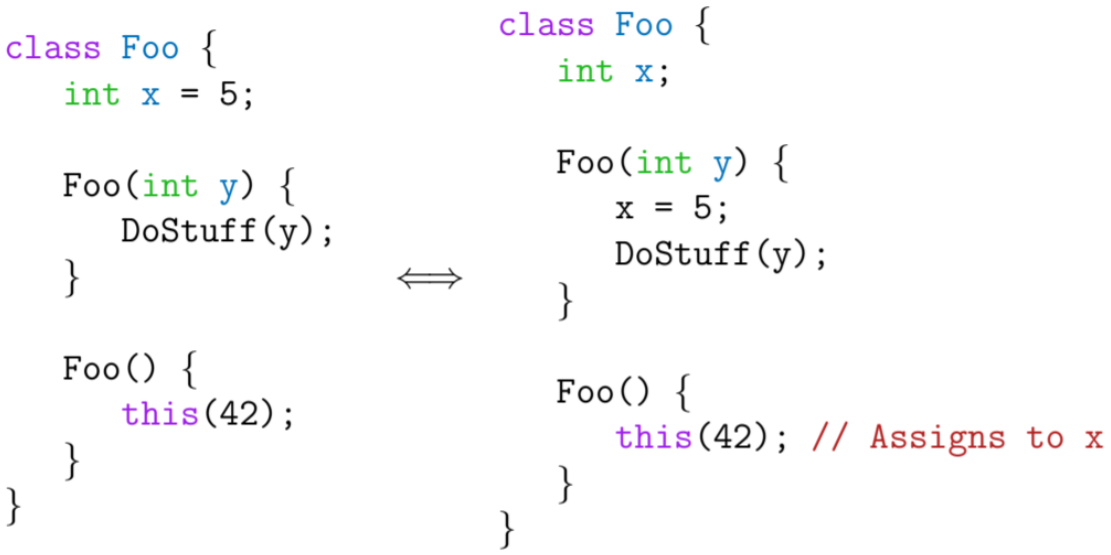
All classes have constructors. If you do not explicitly define one, Java will give your class the **default constructor**:

```
public class Foo {
    public Foo();
}
```

You can define multiple **overloaded** constructors, and they can use each other (although the syntax is a bit odd):

```
public class IntList {
    public IntList(int head, IntList tail) {
        this.head = head; this.tail = tail;
    }
    public IntList(int head){
        this(head, null); // This calls the first constructor
    }
}
```

Instance variable initializations are moved inside constructors that don't start with `this(...)`:



Summary of Java vs Python

So this is a class, its constructor, and some instance and static variables and methods:

```
class Foo:
    """Here is some Python code that creates a class called Foo."""
    def __init__(self,...):
        ...
    def f(self,...):
        ...
    y = 21
    @staticmethod
    def g(...):
        ...

aFoo.f(...)
aFoo.y
Foo(...)
self
```

```
/** Here is some Java code that creates a class called Foo. */
class Foo {
    Foo(...) {
        ...
    }
    int f(...) {
        ...
    }
    static int y = 21;
    static void g(...) {
        ...
    }
}

aFoo.f(...);
aFoo.y;
new Foo(...);
this;
```

As you can see, while coding in these languages feels different, they really are quite similar under the hood.