

Public, Private, Static and all that ...

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University
Semester 1, 2006

Controlling Client Access

Often we want to *limit* the ways in which clients can access and manipulate objects. For example, some methods of a class may only be intended for by other methods of the class — that is, they are “helper” functions.

As an example in Assignment 3, you may want a function which doubles a *row* of an image:

```
ArrayList<Pixel> doubleRow(ArrayList<Pixel> row)
```

This may be used *internally* to help define the method to double the scale of an image, but you don't want to make this function available to the client.

The Interface to a Class

In the **data directed design** approach, a key step was:

Identify the Basic Operations on Objects

These **basic operations** become the **methods** of the class.

Users of such objects are often called **client classes**.

Client classes access and manipulate objects through the methods...

... but they may also be able to directly access and modify the **fields** of objects.

Controlling Client Access

Almost always we want to stop clients directly accessing the **fields** of an object.

Why?

- Because how we choose to **represent** the data is our choice, not the clients.
- In the design process we identified the operations that were needed (and made sense) on objects of this class. That is, the *methods* of the class must be sufficient for all proper uses of the object.
- Giving clients access to the fields means they can circumvent the proper use of the object (via the methods).
- Sometimes, controlling access to fields is a security issue.

An Example

In a recent lecture and prac class we looked at a simple bank account class, `BankAccount.java`.

The operations we identified were

- Create a new account, `BankAccount(Double initialBalance)`
- Get the current balance, `Double getBalance()`
- Make a deposit, `void deposit(Double amount)`
- Make a withdrawal, `void withdraw(Double amount)`

so we expect that clients will use *only those methods* to work with bank accounts.

Still not convinced?

Let's extend the `BankAccount` class to include a record of all transactions.

Each time a deposit is made, record the amount, and each withdrawal is recorded as a negative quantity.

```
ArrayList<Double> transactions;
```

We also want a method to check that the current balance is consistent with the transaction record.

(See `OpenAccount.java`)

Will that stop the fraud? No, if those naughty programmers can change the `balance` field, they can also change the `transactions` field.

(See `Naughty.java`)

However...

There is nothing stopping clients accessing the `balance` field **directly**!

What's wrong with that?

Unscrupulous programmers can defraud the bank by changing their account balance without going through the proper channels (deposit and withdrawal).

Public and Private

Most programming languages provide a means of **controlling which aspects of a module or class are accessible to client classes**.

Java has “access modifiers”: `public` and `private`.

If a field or method declaration is preceded by `private`, it cannot be accessed or modified by client classes.

If a field or method declaration is preceded by `public`, it can be accessed and modified by **all** other classes.

(If we don't specify `public` or `private`, they may be accessed and modified by all other classes in the same directory.)

Make All Fields Private

It is considered good practice to make all fields private. This ensures that the objects of the class are only accessed and manipulated by the methods provided.

Reworking the bank account example (again), we see that making the `balance` and `transactions` fields `private` (and the methods `public`) **prevents** improper access and manipulation.

(See `SecureAccount.java`)

Static

(Treat this as an advanced topic — it will not be in the final exam.)

In earlier lectures I stressed the point that in Java:

An **object** has *its own copies of the methods and fields of the class*.

For example,

```
BankAccount jimsSaving = new BankAccount();
jimsSaving.deposit(2000);
jimsSaving.withdraw(500);
System.out.println(jimsSaving.getBalance());
```

Make “Local” Methods Private

In the PPM example mentioned above, we don't want clients to have access to the `doubleRow` method because it's a helper function only intended for internal use.

Note also that it is not one of the operations that would be identified as a **basic operation on PPM images**.

By making the method `private` we prevent clients from calling it:

```
private ArrayList<Pixel> doubleRow(ArrayList<Pixel> row)
```

Also, when client programmers see that the method is private, they know that they can ignore it completely.

I lied...

We have written things like

```
Math.sqrt((x - p.xCoord())*(x - p.xCoord())) ...
Math.cos(theta) ...
```

but `Math` is a **class**, not an object. (See `java.lang.Math`.)

We didn't say

```
Math thing = new Math();
thing.sqrt(...
```

What's going on?

The Truth

(According to Clem...)

In an earlier lecture I also made the point that Java classes are like Haskell **types** and also like Haskell **modules**.

Sometimes we use **modules** to group together logically related functions and operations. In that case they don't correspond to abstract types in any sense.

Since Java conflates (confuses?) the two ideas, we have to use **classes** for both purposes.

All the methods in `java.lang.Math` are declared `static` so we write things like

```
Math.sqrt(...)
```

We **never** write

```
Math thing = new Math();
```

(In fact `java.lang.Math` doesn't have any constructors, which makes sense.)

Find some other Java library classes like this — there are plenty of them.

(In tomorrow's lecture, we will develop our own module-style class.)

The `java.lang.Math` library is a good example.

It doesn't represent a type — it is a collection of mathematical operators.

In that case, *it doesn't make sense to construct an object of class `Math`*.

We want a way to refer to the methods of these module-like classes.

Putting `static` in front of the declaration of methods and fields of a class means that they are associated with the `class`, not an object.

To refer to a `static` method, we use the `class` name, not an object.