Lambda calculus

- where does a function end?

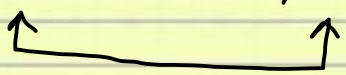
e.g. $\lambda x.\ a\ b\ c$

This appears to be a function whose body is $a\ b\ c$

but $(\lambda x.\ a)\ b\ c$ is the application of the function $\lambda x.\ a$ to b (and then c)

So be careful when functions are arguments

$$(\lambda x.\ x\ b\ c)\ (\lambda x.\ x)$$



carry the parentheses with the function

$$= (\lambda x.\ x)\ b\ c$$

and <u>not</u> $\lambda x.\ x\ b\ c$

Even clearer is

$$(\lambda x.\ x\ b\ c)\ \lambda y.\ a\ b$$

$$= (\lambda y.\ a\ b)\ b\ c$$

not $\lambda y.\ a\ b\ b\ c$

Second point is the use of meta-variables (rather like names)

$$(\lambda x.\ x\ a)\ (\lambda x.\ x)$$

$$= (\lambda x.\ y)\ a$$

$$= a$$

I could have written $(\lambda x.\ x\ a)\ B$ where B names the expression $(\lambda x.\ x)$. Could then write $B\ a$

Brief overview of Java.

- roots in C, C++, Smalltalk
- syntax comes from C

Object-orientation adds to C:

1. encapsulation
2. inheritance
3. polymorphism (late binding of methods)

1. Encapsulation
   - enclose both variables (fields) and functions (methods) with a syntactic construct — the class
   - control access from outside the class boundary

```
class C {          <- field
  int x;
  void f() {...}   <- method
}
```

Conventionally access is controlled to make all data (fields) private and all methods public.

```
public class C {
   private int x;
   public void f() {...}
}
```

An instance of class is an object. Classes define types as well as providing encapsulation.

We can declare a object reference with

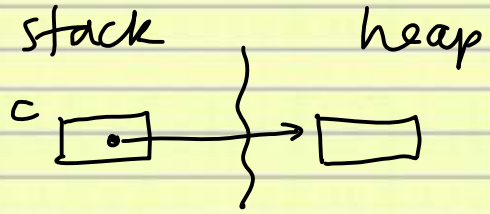
```
C c; // c refers to a non-existant object
```

(C: class, c: variable)

We initialize with operator new

```
C c = new C();
```

This sets c to refer (point) to an object created on the heap.



Access: c.x won't compile since x is private
c.f() will compile since f is public

Direct access to fields within methods:

```
class C {
private int x;
public void f() {
    ... x ....
}
}
```

direct access

We can add accessor functions to control access to private data from outside the class:

```
public class C {
    private int x;
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
}
```

inspector / reader

writer / mutator

```
C c = new C();
c.setX(42);
c.getX() => returns 42
```

We can have both accessors, either one or none.

* public members can be accessed from anywhere
* private members can only be accessed directly in the body of a method of the same class

If we omit public/private we get "package" access — any method in any class in the same package can access any member of the class.

class Car is a subclass of Vehicle

Subtype principle (substitution principle):
An object of a subtype may be used where an object of a supertype is expected.

e.g.
```
Car c = new Car();
Vehicle v = new Vehicle();
v = c;      OK upcasting
c = v;      NOK downcasting
c = (Car) v;  NOK - class cast exception
```