public class Indexed<Type<Identity> I, Type V> {

public Variable<V> access(I index);

V get(I index) {

return access(index).get();

}

void set(I index, V value) {

access(index).set(value);

}

}

interface Type

- name

- variable number of types that compose the type

- explicit composition (constituents) vs. ducks

class Variable<Type T> {

T get();

void set(T value);

T getAndSet(T value) {

T oldValue = get();

set(value);

return oldValue;

}

boolean compareAndSwap(T oldValue, T newValue) {

if (oldvalue == get()) {

set(newValue);

return true;

} else {

return false;

}

}

Type<T> Type;

readonly boolean Mutable;

}

class Property extends Variable {

Type<T> DeclaringType

T,boolean peek();

void poke(T value);

boolean readonly Idempotent = false;   
}

interface Variable<Type<Incrementable> T> {

T preIncrement() {

blindIncrement();

return get();

}

T postIncrement() {

T value = get();

blindIncrement();

return value;

}

void blindIncrement();

T preDecrement() {

blindDecrement();

return get();

}

T postDecrement() {

T value = get();

blindDecrement();

return value;

}

void blindDecrement();

}