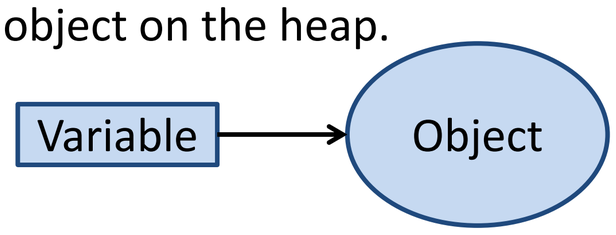
Controlling Aliasing with *Aliasing Contracts*

Janina Voigt, Alan Mycroft
Computer Laboratory, University of Cambridge

What is Aliasing?

In modern OO programming languages, an object reference does not contain the object itself, but the address of the object on the heap.



For this reason, multiple variables can refer to the same object at the same time. We call this ***aliasing***.

A change in one variable impacts all others:

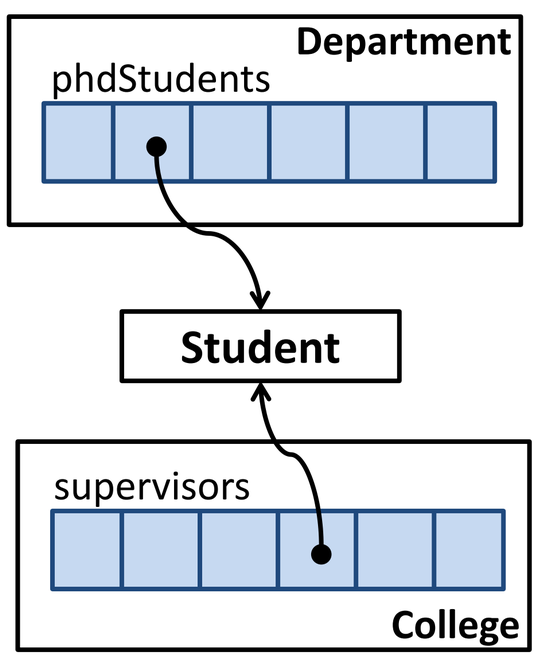
```
Integer x = 5;
Integer y = x;
y = y + 10
print(x + ",," + y);
```

This prints ***15, 15***.

Aliasing for *Sharing*

We need aliasing to implement many common progamming idioms.

Aliasing allows us to share one object between several parts of the system. This is important, for example, when one object can belong to two separate collections:



When Aliasing Goes Wrong

Aliasing can break ***encapsulation***.

The following example is taken from JDK 1.1.1. A particular class records its *signers*, the entities that have digitally signed the implementation.

```
class Class {
  private Object[] signers;

  Object[] getSigners() {
    return signers;
  }
}
```

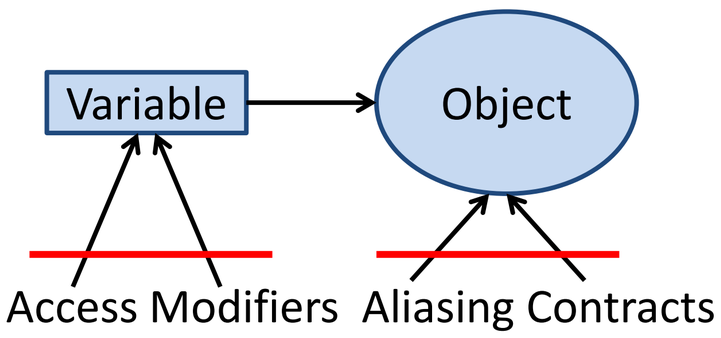
Because `getSigners` returns a reference to the internal `signers` array, any client can now modify `signers` directly, without the `Class`' knowledge.

This is a common problem when getters return references to internal objects.

Aliasing Contracts – Object Encapsulation

In the example above, we saw that making `signers private` was not enough to protect it. Access modifiers like `private` and `public` protect only the variables, not the objects to which they point.

The aim of aliasing contracts is to protect the objects themselves.

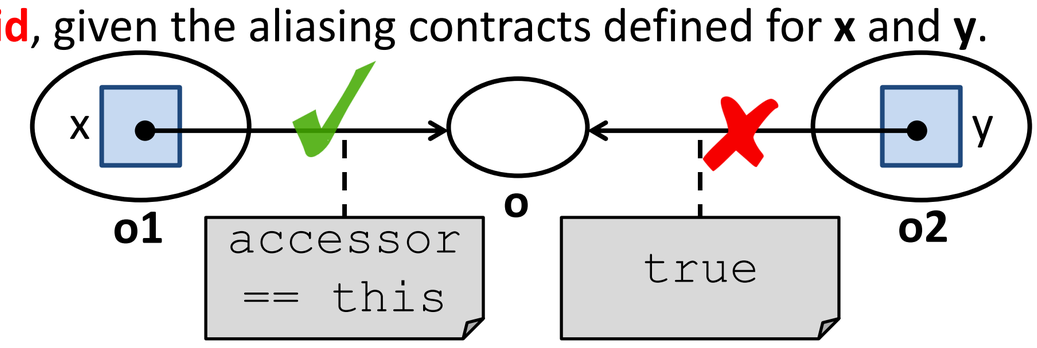


Overview of Aliasing Contracts

- Aliasing contracts specify ***preconditions*** for accesses to an object.
- In the code, contracts are declared on ***variables*** and apply to the ***objects*** to which the variables point.
- Since multiple variables may point to the same object, one object can have multiple contracts.
- Whenever an object is accessed, ***all*** of its contracts must be satisfied, or the access is illegal.

An Example

We have two objects, **o1** and **o2**, with fields **x** and **y**, which both point to the same object, **o**. The tick and cross show which accesses to **o** are **valid** and **invalid**, given the aliasing contracts defined for **x** and **y**.



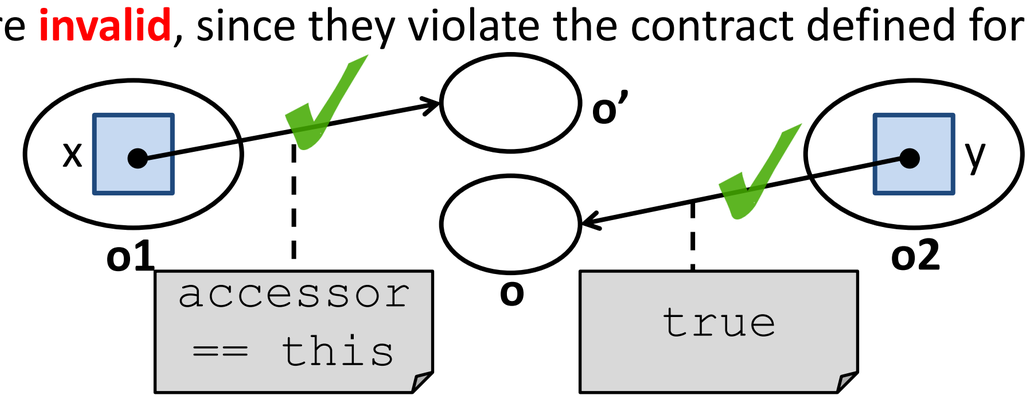
Both **x** and **y** declare aliasing contracts:

- `accessor == this` means that **o1** expects that it (`this`) will be the only object accessing **o**. The keyword `accessor` refers to the object making the access.
- `true` means that **o2** imposes no preconditions on accesses to **o**.

Whenever **o** is accessed, both of these contracts are evaluated. The access is only legal if they are ***both*** satisfied.

Contracts are evaluated in the context of the object which declares them. This means that when we evaluate the contract of **x**, `this` refers to **o1**; `accessor` refers to the object making the access.

In the example above, accesses to **o** from **o1** are **valid**; accesses coming from **o2** are **invalid**, since they violate the contract defined for **x**.



Aliasing contracts depend on the ***dynamic*** aliasing structure of the program. If we reassign **x** to point to another object, accesses to **o** from **y** become **valid**.