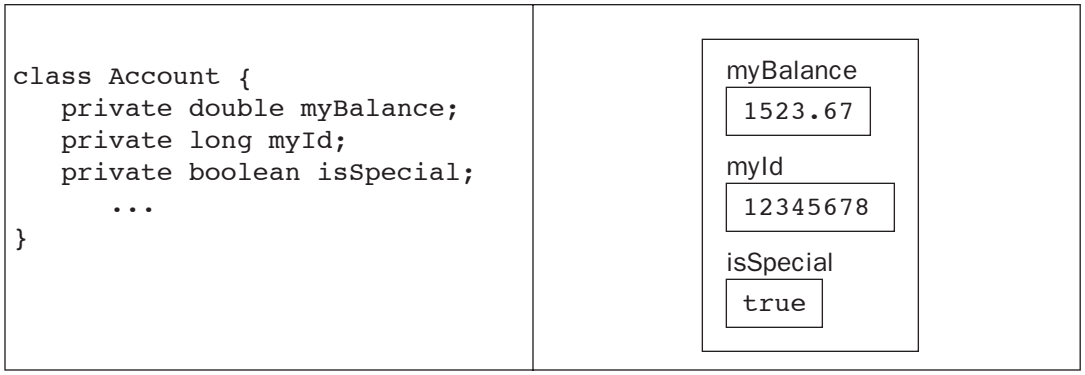
Boxes and Arrows

There are two kinds of variables in Java: those that store primitive values and those that store references. Primitive values are values of type long, int, short, char, byte, boolean, double, and float. References are used to access objects. A box is a good metaphor for either kind of variable.

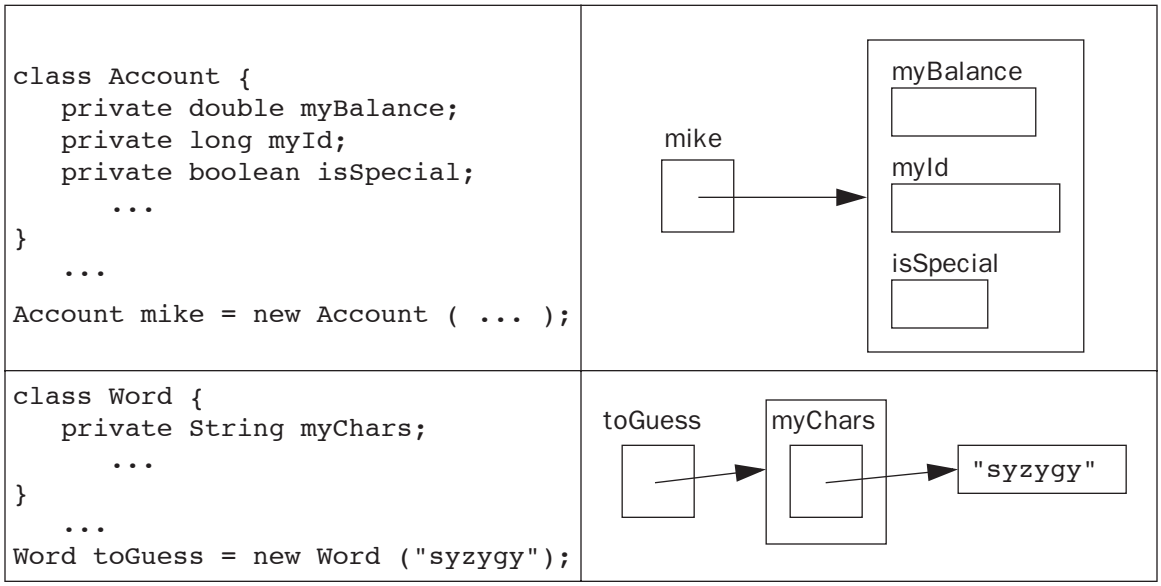
Drawing boxes for primitive variables is easy. The contents of the box is the stored primitive value. A few examples appear below.

an int variable | a boolean variable | a double variable | a char variable

An object is drawn as a box that contains its data members, for example:



Finally, a reference to an object (returned by new) is drawn as an arrow:

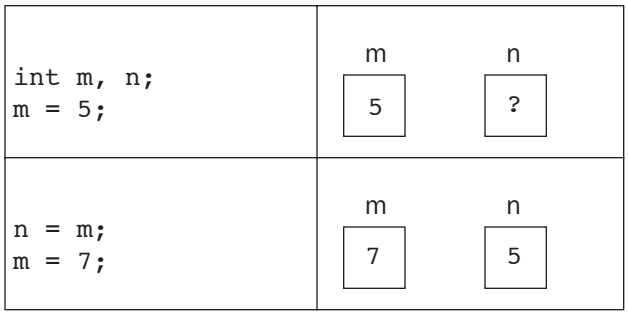


box

A reference-valued variable is sometimes referred to as a *pointer*.

How the assignment statement relates to boxes

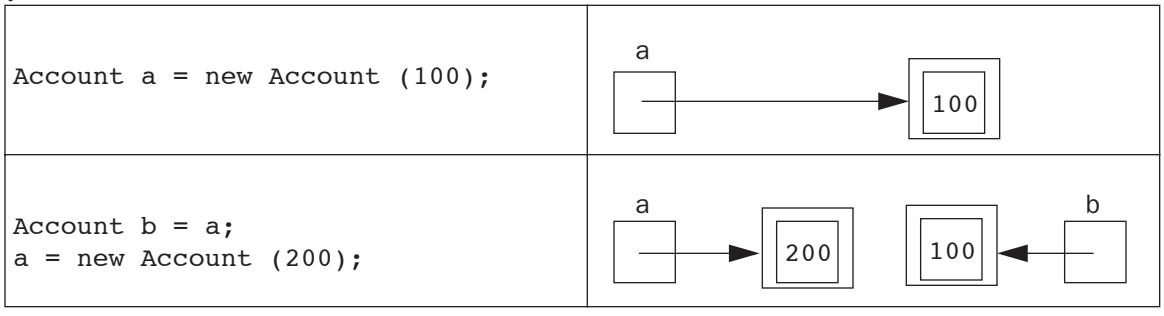
An assignment statement merely copies the value in one box to another. When the box contains a reference—an arrow—two references to the same object result. Below are some sequences of assignment statements, along with their box-and-arrow representations. We start with int variables:



In almost all programming languages, the effect is the same: the value 5 in m gets copied into n, then the value 7 gets copied into m, replacing the 5 that it formerly contained.

Now consider an Account class with a single data member named myBalance and three methods, a constructor, a deposit method, and a withdraw method.

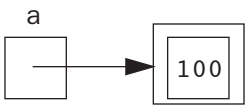
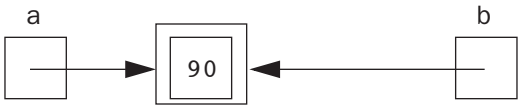
```
public class Account {
   public Account (int initAmount) {
      myBalance = initAmount;
   }
   public void deposit (int amount) {
      myBalance += amount;
   }
   public void withdraw (int amount) {
      myBalance -= amount;
   }
   private int myBalance;
}
```



Here, the reference value initially stored into a—namely, the arrow—is copied into b. After the assignment statement, a and b contain references to the same account. A new reference value is then created via the new operator, and that reference is stored into a, replacing a's former contents.

box

Let's try another one.

<table>
<tr><td><pre>Account a = new Account (100);</pre></td><td> 
</td></tr>
<tr><td><pre>Account b = a;
a.withdraw (10);</pre></td><td> 
</td></tr>
</table>

A reference to a new account is stored into a. That reference (the arrow) is copied into b; both variables now contain references to the same account. The withdrawal from that account is visible either through a's reference or through b's.

In C (without structs) and Scheme, we have essentially the same behavior. Code segments that have the effect of the example just discussed appear below.

C	Scheme

`int a[1] = {100};`
`int* b = a;`
`a[0] -= 10;`
	`(define a '(100))`
	`(define b a)`
	`(set-car! a (- (car a) 10))`

Parameter passing

All method parameters in Java are passed *by value*, which means they are copied into temporary variables in the function. As with the assignment statement, the copying of a reference involves only the reference itself, not the object referred to. (Again, this is the same behavior as in C and Scheme.)

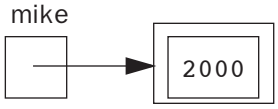
Suppose we supply an additional method for the Account class:

```
public void make900 (Account a) {
   a = new Account (900);
}
```

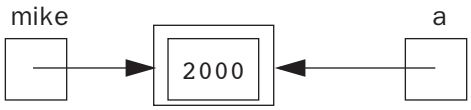
Consider now the effect of the two assignment statements

```
mike = new Account (2000);
make900 (mike);
```

The first sets up a reference to an account with a balance of $2000

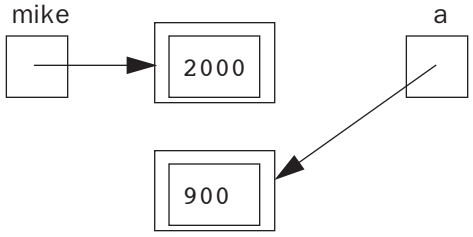


The second statement, a call to make900, creates a copy of the mike reference and stores it in the variable a (which is local to make900).



box

Within make900, a reference to a new account is created and stored into a. The effect is shown below.

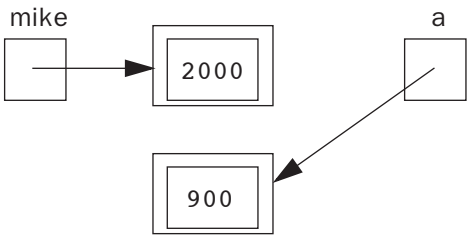
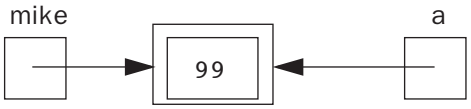


Upon return from make900, the local variable a disappears and the mike reference still points to the account with $2000.

Consider now a method make99 defined inside the Account class:

```
public void make99 (Account a) {
   a.myBalance = 99;
}
```

Contrast the effect of the two sequences of assignment statements:

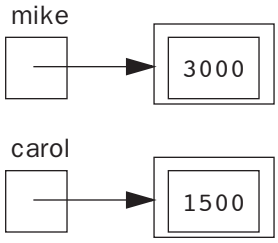
assignment statements	*result immediately before returning from the method*
`mike = new Account (2000);` `make900 (mike);`	
`mike = new Account (2000);` `make99 (mike);`	

For one final example, we consider the following method:

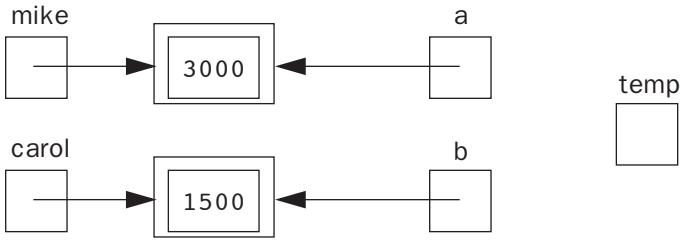
```
public void swap (Account a, Account b) {
   Account temp;
   temp = a;
   a = b;
   b = temp;
}

   ...
Account mike = new Account (3000);
Account carol = new Account (1500);
swap (mike, carol);
```

box

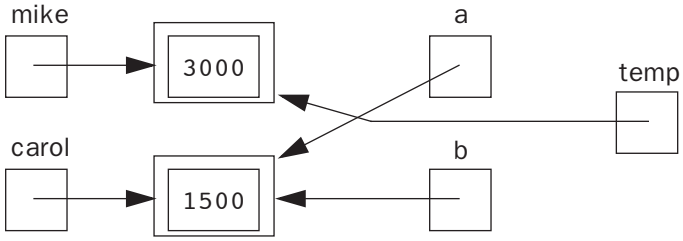
What's the effect of the call to swap? It has no effect on either of the accounts. Here's why. We start with the initialization of mike and carol:



Then we enter the method, copying the argument references to the local variables a and b:



After the first two assignments, we have



The final assignment statement copies a reference to Mike's account into b. Neither the mike variable, the carol variable, nor the corresponding Account objects are modified by the swap method.

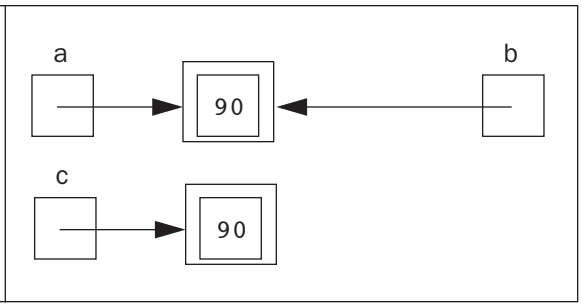
box

Comparisons

A comparison of two references using == determines whether the two things compared are references to the same objects. (This behavior is the same as in C, and the same as what the eq? function provides in Scheme.)

Here's an example.

```
Account a = new Account (100);
Account b = a;
a.withdraw (10);
Account c = new Account (90);
```



After execution of the four statements, comparisons between the reference values would have the results shown below.

expression	*value*	*reason*
`a == b`	`true`	both `a` and `b` refer to the same object
`a == c`	`false`	`a` and `c` refer to different objects, even though those objects both have the same balance
`b == c`	`false`	same reason

Comparison of two *objects* in Java is conventionally done with the `equals` method as shown below.

```
public boolean equals (Account a) {
   return this.myBalance == a.myBalance;
}
```

box