Assignments

An *assignment* statement says that something (the left-hand side) should refer to something else (the right-hand side).

The *syntax* varies (`=`, `:=`, `<-`, `set!`, etc.)

Questions we want to ask:

- What happens *semantically* when we have an assignment?
- What things can and can't be assigned to?
- How do these choices intermix and relate to other concepts in PL design and implementation?

Variable Model

What does an assignment actually do?

We have two basic options:

- **Value model**: Each variable refers to a single value.
  Assignment means *copying* from the r.h.s. to the l.h.s.
  This is the default in C/C++ and SPL.
- **Reference model**: Each variable refers to an object in memory.
  Assignment means changing the l.h.s. to reference the same thing as the r.h.s.
  This is the default in Scheme and many more modern languages.

What do these options remind you of?

Mixing Value and Reference Models I

In Java, *primitive types* (`int`, `boolean`, etc.) follow the value model, while objects follow the reference model.

For example:

```
int x = 5;
int y = x;
++x; // y is still equal to 5!

ArrayList<String> a = new ArrayList<String>();
ArrayList<String> b = a;
a.add("boo"); // Now a and b BOTH have one element, boo.
```

l-values and r-values

An *l-value* is anything that can appear on the l.h.s. of an assignment.
r-values are defined similarly, and generally include any expression.

Under the *reference model of variables*, usually *names* are the only l-values. (But not always!)

Besides names, l-values might also include:

- Array references, like `A[3] = 10;`
- Operator calls, like in

```
int x, y, z;
cin >> x;
(x < 0 ? y : z) = 5;
```

- Function returns, like in `stack.top() = 20;`

What is the assignment statement itself?

In some languages (for instance SPL), an assignment is just a statement, not an expression.

In some languages (for instance Java), an assignment can be an r-value:

```
int x, y;
x = (y = 5); // Sets y, then sets x.
```

In some languages (for instance C++), it can even be an l-value:

```
int x;
(x = 10) = 15; // x is set to 10 and then to 15.
```


Constants and Immutables

A *constant* is a name whose value cannot be changed.
These are declared with special keywords like **const** or **final**.

An *immutable* is an object whose *state* cannot be changed.
For instance, Java `String`s are immutable but not constant:

```
String a = "a string";
a = "anuther string"; // This is fine.
a[2] = 'o'; // This won't compile, for a few reasons.
```

Mixing Value and Reference Models II

In C++, variables declared *as references* follow the reference model:

```
int a = 5;
int& b = a;
a = 10; // Now b is 10 too!
b = 15; // Now a is 15 too!
```

Here we might say that `b` is an *alias* for `a`.

C++ reference variables are clearly not *immutable*, but they are *constant*:

```
int a = 5, b = 6;
int& c = a;
c = b; // Now a and c are both 6.
b = 7; // This still ONLY changes b.
```


Clones

Sometimes we really do want to make copies, even under the reference model of variables.

Java objects that implement `Cloneable` allow this:

```
ArrayList<String> a = new ArrayList<String>();
a.add("hello"); a.add("everybody");
ArrayList<String> b = a;
ArrayList<String> c = a.clone();
a.set(0,"goodbye");
/* Now a and b have ["goodbye", "world"]
 * but c is still ["hello", "world"]. */
```


Class outcomes

You should know:

- The two variable models, and what their differences are.
- What is an alias? What is a clone?
- What are l-values and r-values?
- How do C++ and Java allow us to mix the value and reference models?

You should be able to:

- Trace program execution using the value and reference model of variables.