Assignments

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

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

Questions we should 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 Values References 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"); // a and b BOTH have one element, boo.
```

l-values and r-values

l-value: anything that can be on the l.h.s. of an assignment.
r-value: anything that can be on the r.h.s. of an assignment.

r-values usually include any expression.

l-values can be:

The assignment statement

Depending on the language, an assignment can be:

- Just a statement
- An r-value
- An l-value

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 Values and References 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"]. */
```


Types of Variables

A *type* is a tag on some data in a program that indicates what it means or how it can be used.

Types can be *built-in* (e.g. `int`, `char`, ...)
or *user-defined* (e.g. with `class`, `enum`, `typedef`, ...)

Types can be *declared* (C, C++, Java, Ada, ...)
or *implicit* (*inferred*) (Scheme, Ruby, Perl, Haskell, ...)

Type Safety

Types provide documentation and help ensure data gets used correctly.

Type safety is a mechanism enforced by the compiler or interpreter to ensure that types are not used in an incorrect or meaningless way.

Languages with type safety are less prone to errors and exploits.
Nearly every modern language has some type safety.
Some languages allow explicit overwriting of type safety checks.

Dynamic vs Static Typing

Where is type information stored?

- **Dynamic Typing**:
  Types are stored with data objects, at run-time.
  Makes sense for interpreted languages.

- **Static Typing**:
  Types stored with symbols, and *inferred* for expressions, at compile-time.
  Very useful in compiled languages.

Type inference

This refers to the automatic determination of an expression's type.

- **Simple example**: `5 + 3`
  has type `int` because 5 and 3 are both `int`s.
- **More difficult**: `5 + 3.2`
  Is this a `double` or `int`?
  Depends on rules for *type promotion*/*coercion*.
- **Totally crazy**: Some languages like ML infer the types of all variables, arguments, and functions based on how they are used.
  Type consistency is ensured at compile-time!

What gets a type?

Constants or *literals* such as `-8`, `'q'`, `"some string"`, and `5.3` will all have a type.
Expressions will generally have the type of whatever value they compute.

- **Names**: Only have a fixed type in *statically-typed* languages.
- **Functions**: Type is determined by number and types of parameters and type of return value.
  Can be thought of as pre- and post-conditions.
  May be left unspecified in dynamically-typed languages.
- **Types**: Do *types* have type? Only when they are first-class!

Type Checking

Type checks ensure type safety.
They are performed at compile-time (*static*) or run-time (*dynamic*).

- **Dynamic Type Checking**: Easy! Types of arguments, functions, etc. are checked *as they are applied*, at run-time. Every time an object is accessed, its type is checked for compatibility in the current context.
- **Static Type Checking**: Type safety is ensured at compile-time.
  The type of every node in the AST is determined statically.
  Some level of *type inference* is always necessary.
  Often, *type declarations* are used to avoid the need for extensive inference.

Class outcomes

You should know:

- The two variable models, and what their differences are.
- What are l-values and r-values?
- What is an alias? What is a clone?
- How do C++ and Java allow us to mix the value and reference models?
- The benefits of type safety in programming languages.
- The differences between static and dynamic typing.
- The meaning of type inference.

You should be able to:

- Trace program execution using the value and reference model of variables.
- Demonstrate dynamic and static type checks for an example program.