In this class, we will extend JAKARTA_SCRIPT with the basic language primitives that we need to support object-oriented programming. We will stay faithful to the semantics of objects in JavaScript while restricting the new language features to the bare minimum. Certain features of JavaScript’s object system, such as prototype objects, will not be supported directly by our language extension. Instead, we will study how we can implement the essential features of object-oriented (OO) programming languages by using only the very simple primitives of our minimal language extension.

A Simple Untyped Language with Objects

We start from the language that we discussed in class 18 and add object literals and field dereference operations. We first describe the semantics of the new language constructs informally.

An object literal is a comma-separated sequence of field names with mutabilities (const or var) and initialization expressions surrounded by braces:

\[
\{ \text{mut}_1 f_1 : e_1, \ldots, \text{mut}_n f_n : e_n \}
\]

When an object literal is evaluated, we create an object. An object is a mapping from field names to values. Evaluating an object literal creates an object that maps the fields \( f_i \) to the values \( v_i \) obtained from evaluating the initialization expressions \( e_i \). The created object is stored in memory and the object literal itself evaluates to the address of that memory location.

To access the value of a field \( f \) in an object, we use field dereference operations, \( e . f \). Here \( e \) must evaluate to an address \( a \) of a memory location that contains an object. The operation \( e . f \) then evaluates to the value of the field \( f \) in the object stored at address \( a \).

Fields of objects are mutable if they are prefixed with the mutability var otherwise they are immutable. Mutable fields can be reassigned new values using assignment expressions of the form \( e_1 . f = e_2 \). The effect of such an assignment is that the object at the memory location given by \( e_1 \) will be modified by setting its field \( f \) to the value obtained from \( e_2 \).
**Aliasing.** Since objects are referenced with an extra level of indirection through an address, two program variables can reference the same object. We refer to such a situation as *aliasing*. With mutation, aliasing is now observable as demonstrated by the following example:

```javascript
const x = { var f : 1 };
const y = x;
x.f = 2;
y.f
```
This program will evaluate to 2 because `x` and `y` are aliases (i.e., reference the same object). The interaction between aliasing and mutation makes programs more difficult to reason about and is often the source of subtle bugs.

**Abstract Syntax.** For the time being, we remove types from our language. We will discuss typing issues related to objects in the next class. The abstract syntax of our new language is given by the following grammar:

\[
\begin{align*}
n & \in \text{Num} \quad \text{numbers (double)} \\
b & \in \text{Bool} ::= \text{true} | \text{false} \quad \text{Booleans} \\
a & \in \text{Addr} = \mathbb{N} \quad \text{addresses} \\
x & \in \text{Var} \quad \text{variables} \\
f & \in \text{Fld} \quad \text{field names} \\
v & \in \text{Val} ::= n | b | a | \text{function } p(x) e \quad \text{values} \\
le & \in \text{LVal} ::= *a | a.f \quad \text{location values} \\
e & \in \text{Expr} ::= x | v | e_1 \text{ } bop e_2 | uop e_1 | e_1 ? e_2 : e_3 | \quad \text{expressions} \\
mut & x = e_d ; e_b | e_1 (e_2) | e.f | \{ \text{mut}_1 f_1 : e_1 , \ldots , \text{mut}_n f_n : e_n \}
\end{align*}
\]

\[
\begin{align*}
le & \in \text{LExpr} ::= x | e.f \quad \text{location expressions} \\
bop & \in \text{Bop} ::= + | \star | \& \& | \mid \mid = \quad \text{binary operators} \\
uop & \in \text{Uop} ::= \star \quad \text{unary operators} \\
p & ::= x | \epsilon \quad \text{function names} \\
mut & \in \text{Mut} ::= \text{const} | \text{var} \quad \text{mutabilities}
\end{align*}
\]

We introduce *location expressions*, `le ∈ LExpr`, as a separate semantic domain. Location expressions are the expressions that we allow to occur on the left side of assignment expressions. Similarly, we introduce *location values* to denote the expressions that *location expressions* in assignments reduce to during evaluation.

In our concrete syntax, we allow mutabilities to be omitted from fields in object literals. The corresponding fields will be considered to have the default mutability `var`. We make this choice in order to be compatible with JavaScript, which does not support `const` fields.
\[
\begin{align*}
\langle M, e \rangle &\rightarrow \langle M', e' \rangle \\
\langle M, \{ \ldots, \text{mut } f : e, \ldots \} \rangle &\rightarrow \langle M', \{ \ldots, \text{mut } f : e', \ldots \} \rangle \\
\langle M, e_1 \rangle &\rightarrow \langle M', e'_1 \rangle \quad e_1 \notin \text{LVal} \\
\langle M, e_1 = e_2 \rangle &\rightarrow \langle M', e'_1 = e_2 \rangle \\
\langle M, lv = e_2 \rangle &\rightarrow \langle M', lv = e'_2 \rangle \\
\langle M, e \}. f \rangle &\rightarrow \langle M', e'. f \rangle \\
\end{align*}
\]

\text{SearchObject} \quad \text{SearchAssign}_1 \quad \text{SearchAssign}_2 \quad \text{SearchGetField} \quad \text{DoObj} \quad \text{DoGetField} \quad \text{DoAssignFld}

Figure 1: New and updated inference rules for the language primitives related to objects

**Small-Step Semantics**

Figure 1 shows the small-step reduction rules for the new language constructs. A memory state \( M \) is now a partial function from addresses to values and object values. We formalize this by introducing a semantic domain of memory contents:

\[
k \in \text{Con} ::= v \mid \{ f_1 : v_1; \ldots; f_n : v_n \}
\]

and define

\[
M \in \text{Mem} = \text{Addr} \rightarrow \text{Con}
\]

Note that the fields of object values in memory have no mutabilities. The type system that we will discuss in the next class will statically ensure that immutable fields cannot be reassigned. Hence, we can erase the mutability information at evaluation-time.

The interesting rules are the do rules. The DoObj rule formalizes the allocation of a fresh memory location to store the result of evaluating an object literal. The DoGetField rule formalizes the semantics of field dereference via memory look-up. Finally, the rule DoAssignFld formalizes the assignment to a field by updating the object at the given memory location.
Objects and Classes

We now show how the essential features of object-oriented languages can be realized using the language primitives that we introduced above. We discuss simple objects, classes and data encapsulation, inheritance, method overriding, calls to super class methods, and open recursion. We use the concrete syntax of JavaScript so that you can easily try out the examples using a JavaScript interpreter.

Simple Objects

We start by introducing simple objects. An object in object-oriented programs encapsulates state and provides methods to access and manipulate this state. We also refer to the state of the object as the representation of the object.

As an example, we implement a object that encapsulates a counter value \( x \). The counter object provides two methods. A method `get` that reads the current counter value, and a method `inc` that increments the counter. We can declare such an object in our language using an object that has two fields holding functions that implement the `get` and `inc` methods:

```javascript
const rep = { x: 0 };
const counter = {
  get: function() { return rep.x },
  inc: function() { rep.x = rep.x + 1 }
};
```

You may wonder why we declared the field `x` that holds the counter value in a separate object, `rep`, instead of adding it directly to the `counter` object. There are two reasons for this. First, if `x` was a field of `counter`, then any client who has a reference to `counter` would be able to modify the state of the counter directly by assigning new values to the field `x`. This would break the OO philosophy of hiding the representation of an object from the clients of the object. Most object-oriented languages provide mechanisms for controlling the visibility of fields. We do not have such mechanisms in place, yet. Hence, we put representation of the `counter` object in another object `rep` that is read and written by the `get` and `inc` methods. Hence, a client of the `counter` object who does not have a direct reference to the `rep` object will not be able to modify the state of `counter` directly.

The second reason why we cannot add the field `x` to `counter` directly is that the methods `inc` and `get` must be able to access `x`. However, we cannot access a field of an object before the object has been created. Most OO languages (including JavaScript) support access to the fields of an object from inside the object’s methods through a special variable called `this` (or `self`). The `this` reference is only bound to the actual object instance after the object has been created. This feature is called open recursion. Instead of building this feature directly into our language, we will later see how to implement it using the language primitives that we already have at our disposal.
Here is a simple client of our counter object that calls `inc` three times and then returns the new counter value:

```javascript
const counterClient = function(c) {
    c.inc();
    c.inc();
    c.inc();
    return c.get();
};
```

Evaluating the `counterClient(counter)` will return 3, as expected.

### Classes and Encapsulation

So far, we have only constructed a single object instance. Constructing objects this way is rather tedious. Moreover, we have not yet achieved actual encapsulation of the state of our counter object, since we still have a global variable `rep` to the counter representation. Hence, clients can still by-pass the methods of the object and access the internal state directly. We will solve both of these problems at once.

Many OO languages are based on `classes`. A class can be thought of as a template that describes how new instances of a specific type of objects can be created. The class mechanisms of most languages tend to be rather complicated because they are often the only language construct for coarse-grain structuring of programs. Hence, classes often provide many features that are orthogonal to their main purpose. We here focus on the basic features provided by a `class`.

We implement a `counter class` as a function that takes the internal representation of a counter object and then returns a counter object instance:

```javascript
const counterClass = function(rep) {
    return {
        get: function() { return rep.x },
        inc: function() { rep.x = rep.x + 1 }
    };
};
```

Using the function `counterClass` we can now define a `constructor` that creates new counter objects on demand:

```javascript
const newCounter = function() {
    const rep = {x: 0};
    return counterClass(rep);
};
```

The scope of the reference to the `rep` object of the newly created `counter` object is now restricted to the body of the constructor. Hence, the clients of `counter` objects can no longer read or modify the counter values directly without calling the `get` and `inc` methods.
Also, note that each counter object has its own internal representation. For example, the following code evaluates to 0 because the counter value of counter2 is not modified by the calls to inc on counter:

```javascript
const counter = newCounter();
const counter2 = newCounter();
counterClient(counter);
counter2.get()
```

**Inheritance**

One of the most important features of OO languages is that they provide an inheritance mechanism to augment objects with additional functionality in a modular fashion.

As an example, we define a new class of reset counter objects. A reset counter provides the same functionality as a counter object. In addition, it provides a method reset that sets the internal counter value back to 0. The following function implements our reset counter class:

```javascript
const resetCounterClass = function(rep) {
    const _super = counterClass(rep);
    return {
        get: _super.get,
        inc: _super.inc,
        reset: function() { rep.x = 0 }
    }
};
```

The function `resetCounterClass` first calls `counterClass` with the given `rep` object to create a counter object `_super`. Then it creates a reset counter object using the get and inc methods of `_super` and adds the reset method. Thus, the reset counter class effectively inherits the get and inc methods from `counterClass`. We refer to the extended class as the subclass and to the class that is extended as the superclass.

The constructor for reset counter objects is almost identical to that of counter objects:

```javascript
const newResetCounter = function() {
    const rep = {x: 0};
    return resetCounterClass(rep);
};
```

**Substitution Principle**

One of the key properties of object-oriented programs is that objects of subclasses can be safely used in any context that expects an object of a superclass. Here, “safe” means “type safe”, i.e., the evaluation of a program in which an
object of a superclass has been replaced by an object of a subclass cannot get stuck.

For example, we can safely call `counterClient` on a reset counter object as in the following program:

```javascript
const counter = new ResetCounter();
counterClient(counter);
counter.reset();
counter.get()
```

This program will evaluate to 0.

We refer to the principle of safe replacement of superclass objects by subclass objects as the *substitution principle*. We will study this principle more thoroughly when we add a type system to our object-oriented version of JAkarTaScript.

### Extending the Internal Representation and Overriding Methods

When we derive a subclass from a superclass we cannot only add new methods but we can also add new fields to the internal representation. Moreover, we can *override* the methods of the superclass with new methods that change the behavior of subclass instances.

As an example, we define a class of backup counter objects. This class extends the reset counter class with a new method `backup` that stores the current counter value in an additional field `y`. The field `y` is add to the internal representation. The backup counter class then overrides the `reset` method so that the counter is set back to the backed up value stored in field `y`:

```javascript
const backupCounterClass = function(rep) {
    const _super = resetCounterClass(rep);
    return {
        get: _super.get,
        inc: _super.inc,
        reset: function() { rep.x = rep.y },
        backup: function() { rep.y = rep.x }
    }
}

const newBackupCounter = function() {
    const rep = {x: 0, y: 0};
    return backupCounterClass(rep);
}
```

### Calling Superclass Methods

When we extend a superclass with new methods or when we override existing methods, it is often useful to implement the new functionality by calling the existing methods provided by the super class.
For example, suppose we want to implement a variant of our backup counter class that backs up the counter value each time `inc` is called. Then, we can implement the new `inc` method using the `backup` and `inc` methods of the backup counter class as follows:

```javascript
const funnyBackupCounterClass = function(rep) {
    const _super = backupCounterClass(rep);
    return {
        get: _super.get,
        inc: function() { _super.backup(); _super.inc() },
        backup: _super.backup
    };
};
```

Note that the call `_super.inc()` will go to the `inc` method defined in the function `backupCounterClass`.

**Open Recursion**

Finally, we consider the problem of open recursion via a self reference `this`. Open recursion is often useful, e.g., to implement one method using calls to other methods of the same class. For example, suppose we want to implement a counter object that provides a `set` method, which can be used to set the counter to a given value. Then we can implement `inc` using calls to `get` and `set` on the same object instance.

There are essentially two ways how we can implement open recursion in our language. First, we can make the function that implements our counter class recursive so that it first creates an object instance where the behavior of the methods is only partially defined. Then it calls itself recursively to “tie the knot”, binding the created instance to `this` to get the intended behavior. However, this approach is rather inefficient because we will end up constructing multiple object instances.

There exists a more efficient solution that closely follows the actual implementation of open recursion in OO languages. In this solution we tie the recursive knot by using the indirection provided by references to the memory and combination with state mutation.

The trick is to change the class function so that it takes a reference, say `_this`, to an object in memory that already provides all the necessary fields. The function implementing the counter class then simply modifies the `_this` object by updating its fields to the appropriate values:

```javascript
const setCounterClass = function(_this) {
    _this.x = 0;
    _this.get = function() { return _this.x });
    _this.set = function(i) { _this.x = i }
    _this.inc = function() { _this.set(_this.get() + 1) }
    return _this;
};
```
The constructor method now simply needs to create a *dummy* object instance that provides the correct fields, initialized with some placeholder values of the appropriate types. The reference to the dummy instance is then passed to the class function, which initializes the fields to the correct values:

```javascript
const newSetCounter = function() {
    const dummy = {
        x: 0,
        get: function() { return 0 },
        set: function(i) {},
        inc: function() {}
    };
    return setCounterClass(dummy);
};
```