SaM Primitives for Object Orientation

- Object allocations need to be done on the heap. SaM provides the `MALLOC` and `FREE` instructions to accomplish this.
  - For `MALLOC`: Size of object taken from TOS; memory address returned on TOS.
  - For `FREE`: Memory address to be freed taken from TOS.

SaM Primitives for Object Orientation

- Object allocations need to be done on the heap. SaM provides the `MALLOC` and `FREE` instructions to accomplish this.
  - For `MALLOC`: Size of object taken from TOS; memory address returned on TOS.
  - For `FREE`: Memory address to be freed taken from TOS.
- Instance methods need to be dynamically dispatched. SaM provides the `JSRIND` instruction for this purpose.
  - The program address of the called method is taken from TOS.
  - The return address of the call site is pushed on TOS.

Simplifying LiveOak-3 Implementation On SaM

- Since we don’t have inheritance, we will simplify the run-time representation of objects.
  - Retain only the data members in the object record.
  - No need for class object and their associated records and pointers.

Simplifying LiveOak-3 Implementation On SaM

- Since we don't have inheritance, we will simplify the run-time representation of objects.
  - Retain only the data members in the object record.
  - No need for class object and their associated records and pointers.

- We will also simplify instance method invocations.
  - No dynamic method dispatch needed, because no new methods to be added or overridden in subclasses. So we will handle it using name mangling of method names in both definitions and invocations.

Simplifying LiveOak-3 Implementation On SaM

- Since we don't have inheritance, we will simplify the run-time representation of objects.
  - Retain only the data members in the object record.
  - No need for class object and their associated records and pointers.

- We will also simplify instance method invocations.
  - No dynamic method dispatch needed, because no new methods to be added or overridden in subclasses. So we will handle it using name mangling of method names in both definitions and invocations.

- This strategy would not generalize to inheritance.
  - Given the LIFO nature of memory, handling class objects is rather cumbersome (but of course possible).
  - If we used this variation, we would extract the correct method name from the class object's method table and dispatch using the `JSRIND` instruction.

Constructors

- Handle object constructors with a wrapper.
- Translate the expression `new C(e1, ..., en)` as follows.
  - The caller does the necessary `MALLOC` and passes that reference as the first parameter to the constructor call.
  - Since there is no `return` statement in the constructor, the caller also takes care of sticking the object reference into the return value slot of the constructor invocation.

Constructors

- Handle object constructors with a wrapper.
- Translate the expression `new C(e1, ..., en)` as follows.
  - The caller does the necessary `MALLOC` and passes that reference as the first parameter to the constructor call.
  - Since there is no `return` statement in the constructor, the caller also takes care of sticking the object reference into the return value slot of the constructor invocation.

```
// space for return value
PUSHIMM 0
PUSHFBR
// create object and
// push reference on stack
MALLOC size-of-C
code for e1
...
code for en
PUSHSP
PUSHIMM (n+2)
SUB
PUSHFBR
JSR C$new // to the constructor
// pop off all parameters except
// for object reference
ADDSP -n
// stick into return value slot of
// dying constructor invocation
STOREOFF -1
POPFBR
```

Instance Variable References

- If `x` is an instance variable of an object of class `C`, then an unadorned reference to `x` within an instance method or constructor is shorthand for `this.x`.
  - This receiver object is always the first parameter of the method, i.e., in offset +1 from the current FBR.
  - The (compile-time) symbol table for class `C` has the offset *ox* for field `x` in the object record.
- Use the following templates to generate code for read and write references.

Instance Variable References

- If `x` is an instance variable of an object of class C, then an unadorned reference to `x` within an instance method or constructor is shorthand for `this.x`.
  - This receiver object is always the first parameter of the method, i.e., in offset +1 from the current FBR.
  - The (compile-time) symbol table for class C has the offset *ox* for field `x` in the object record.
- Use the following templates to generate code for read and write references.

```
// Template for ... this.x ...
PUSHOFF 1
PUSHIMM ox
SUB
PUSHIND // read value from memory
```

Instance Variable References

- If `x` is an instance variable of an object of class C, then an unadorned reference to `x` within an instance method or constructor is shorthand for `this.x`.
  - This receiver object is always the first parameter of the method, i.e., in offset +1 from the current FBR.
  - The (compile-time) symbol table for class C has the offset *ox* for field `x` in the object record.
- Use the following templates to generate code for read and write references.

```
// Template for ... this.x ...
PUSHOFF 1
PUSHIMM ox
SUB
PUSHIND // read value from memory
```

```
// Template for this.x = rval;

PUSHOFF 1
PUSHIMM ox
SUB
code for rval
STOREIND // write value to memory
```