Java’s object model:

- Java pretty much follows the object model of Smalltalk (1970’s)

In Java, all programming is done by creating classes

A class is a structure:

1) The methods of the class ← we will do this in your interpreter

2) The super class of the class ← we will do this

3) The constructors of the class ← optional

4) The static fields (class fields) of the class ← we will do this as well (optional)

5) The instance field names ← we will do this

6) Static initializers

Each of these can be considered an environment (state structure)

public class MyClass extends AClass {

private int field1 = 10;

private int field2;

private static String field3;

public MyClass() { // default, provided if we don’t write a constructor

super();

}

public MyClass(int f1) {

super();

field1 = f1;

}

public MyClass(int f1, int f2) { // overloading a constructor

this(f1); // calls another constructor of the same class

field2 = f2;

}

public int myMethod(int a, int b) {

this.field1 = a + b;

}

public int myMethod(int a, int b, int c) { // overloaded method

this.field2 = a + b + c;

}

If you want to allow overloaded methods, you must lookup methods by both the name and the parameter signature.

public static void myStaticMethod(String s) {

field3 = s;

}

What is the difference between a static and a non-static method?

Non-static methods have a special parameter “this”.

When you create the closure for a non-static method, add an additional parameter “this” to the list of parameters.

Function closure: paramters, body, function to get the state for the function

Everytime we create an instance

MyClass m = new MyClass(3)

we create an object stored in the heap.

An object is a structure:

1) The class of the object (true type, runtime type)

2) All the instance field values (for all fields of this type and all super types)

((x y z) (1 2 3))

class A { class B extends A { class C extends B {

int x = 10 int x = 100; int x = 1;

int y = 20 } int y = 2;

int z = 30 }

void setX(int x) {

this.x = x;

}

}

A a = new A()

A b = new B()

A c = new C()

a.x = 10 b.x = 10 c.x = 10

a.y = 20 b.y = 20 c.y = 20

a.z = 30 b.z = 30 c.z = 30

c.setX(4)

A B C

x y z x x y

c’s object : {C class, { 10, 20, 30, 100, 1, 2}} (not necessarily in this order)

C class: {B class, {x, y}, methods}

We want to make sure that when we typecast an instance, we only see the fields available for that type.

Trick: We want to store the values in the opposite order. (Store the field names in “normal order”, store the values in the instance in the reverse order)

C B A

names: C’s y, C’s x, B’s x, A’s z, A’s y, A’s x

values: A’s x value, A’s y value, A’s z value, B’s x value, C’s x value, C’s y value

(x y x z y x)

y -> x -> x -> z -> y -> x

| |

C B ^

|

A’s fields

A-list = (cons ‘z (cons ‘y (cons ‘x ()))))

B-list= (cons ‘x A-list)

C-list = (cons ‘y (cons ‘x B-list))

A: (z y x)

c : ~~(10 5 7 4 2 9)~~  (9 2 4 7 5 10)

b: (8 5 7 4)

B: (x z y x)

C: (y x x z y z)