CS61B, Spring 2003 **Discussion #2** **Amir Kamil**
UC Berkeley **2/3/03**

Topics: Static vs. Instance Methods, Pointers, Control Structures

1 Static vs. Instance Methods

As a preliminary to our discussion on static methods, forget for a few moments that you know anything about object oriented programming. Pretend that you never did it in CS61A, and that we have not even mentioned the word 'object' in CS61B.

1.1 Functions in Java

As in Scheme, it is possible to write functions (also called *static methods*) in Java. (Of course, since we don't know what object oriented programming is, it better be possible or else we wouldn't be able to write anything.) The anatomy of a Java function is as follows (disregard the access modifier):

```
static void <return type> <function name>(<type1> <var1>, ... , <typeN> <varN>) {
  <function body>
}
```

There is one immediately obvious difference between Scheme and Java (besides the different syntax). Java is a *stronly-typed* language. This means that all variables have a type, and only values that correspond to a variable's type can be assigned to that variable. This is why each parameter in a function has a declared type. In addition, each function must also declare its return type, and can only return values that correspond to that type. (Here I say *correspond*, because it is not necessarily the case that the types must be the same. This is a subtopic of inheritance).

One more difference between Scheme and Java is that a value must be explicitly returned from a Java function using `return <value>`. Java does not implicitly return the value of the last statement in the function.

As an example, let's write a function in Scheme and Java that compute the same thing. Let's keep it simple by writing the `factorial` function. In Scheme, it is:

```
(define (factorial x)
  (if (= x 0) 1 (* x (factorial (- x 1)))))
```

The same function in Java is:

```
static int factorial(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * factorial(x - 1);
  }
}
```

Note the differences that we mentioned above in this function.

Of course, like Scheme, Java is not purely functional since they both allow side-effects. The Java equivalent of global variables is as follows:

```
static <type> <var> = <value>;
```

As an example, if we wanted to define a global variable called `count`, in Scheme it would be

```
(define count 0)
```

and in Java it would be

```
static int count = 0;
```

The initialization is optional in Java. Without it, a variable is assigned a default value (in this case it would still be 0). Then it is possible to assign to the global variable from within a function.

There is one useful feature that Java has but Scheme does not (as far as I know). In Java, you can (and must, actually) arrange functions and global variables in *modules*. A module is defined as follows:

```
class <module name> {
  <global variables and functions>
}
```

Now in order to access a module's *member* from a different module, the syntax `<module name>.<var>` or `<module name>.<function>(<args>)` must be used, where `<module name>` is the name of the module in which the member is defined. For example, if a function `foo()` was defined in module `A`, in order to call it from module `B`, we have to say `A.foo()`. There are multiple benefits to this arrangement, one of which is that we can have multiple functions of the same name but in different modules and be able to use all of them. As an aside, Java does allow us to write functions with the same name in the same module as long as they take in different arguments. Then when we call a function, the arguments determine which one to use.

Let's digress a bit for a moment. How do you run a Java program? In Scheme, we just typed in the function we wanted to call in the interpreter, but we don't have a Java interpreter. In fact, Java programs are *compiled* and then run from a command line as `java <module name>`. Note that in order for a change in a Java source program to reflect when you run it, the program must be recompiled after the change.

The problem with using the above method to run programs is that there is no way for the Java runtime to know which function you want to call. This wasn't a problem in Scheme since we just called the function we wanted from the interpreter. So there are two ways to deal with this problem. One is to force the user to specify which function to call from the command line as `java <module name>.<function>(<args>)` or something similar. Unfortunately, this requires the user to know something about the inner workings of the program, specifically which function to call. The second solution is to use a convention of calling the same function regardless of which program is run. For example, whenever a user types in `java <module name>` in the command line, we can always call the function `<module name>.foo()`. This is the solution Java uses, except the function called must have signature (access modifier is irrelevant):

```
static void main(String[] args) {...}
```

Any parameters to the program are placed in the `args` array. For example, if the user types in `java <module name> foo bar baz`, then the resulting array would be `{"foo", "bar", "baz"}`, and `main()` would be called with that argument.

That is in essence functional programming in Java. Unfortunately, there are limits to what we can accomplish using only functional programming, especially when it comes to data structures. The only data structure we have at our disposal is arrays, which suffice in some cases but not in others. Specifically, if we wanted Scheme-like lists, there is no way we can implement them using arrays. So we must extend our language to allow us this functionality.

1.2 Objects

Thankfully, Java contains a type of structure called an *object*. An object is a conglomeration of data (called *fields* or *instance variables*), which can be primitive values (such as `int`s or `float`s) or references to other objects. An object definition is similar to a module definition:

```
class <object type> {
  <variables>
}
```

However, variable definitions in an object are slightly different than in a module. In an object, the definition does not contain the keyword `static`. So it is:

```
<type> <var> = <value>;
```

Again, the initialization is optional, and without it the variable is assigned a default value. A particular example of an object type is called an *instance*, and each instance is independent of all others. So instance `A` has its own values of the variables `value` and `next`, and instance `B` has its own independent values of the same variables.

What is the default value of a reference variable? In Java, there is a special reference `null` that signifies a lack of an object, similar to Scheme's `nil`. This is the default value for a reference variable.

As an example, let's define an object corresponding to a Scheme-type list. A Scheme list consists of a pair, with the first member of the pair being an arbitrary value and the second a reference to another list. Since Java is typed, we have to constrain what values our list can contain, so let's store `int`s. Then our list definition would be:

```
class SchemeList {
  int value;
  SchemeList next;
}
```

Yes, it is allowed for an object to contain a reference to another object of the same type. Like in Scheme, a `null` value for the `next` field signifies the end of a list.

The ability to define objects, of course, would be useless without the ability to create them. To create an object in Java, we use the syntax `new <object type>()`. For example, in order to create a `SchemeList` and assign it to a variable `x`, we would say:

```
SchemeList x = new SchemeList();
```

Again, being able to create an object is useless unless we can access and modify its fields. The way to do this is `<object reference>.<field>`. Note that an actual reference is required on the left side of the dot operator. So it is legal to say `x.value` but not legal to say `SchemeList.value`, since `x` is an actual reference while `SchemeList` is just a type. Note that this is in contrast to how we access a module's member. Then if we have a reference `y` to a different `SchemeList` object, `y.value` is completely independent from `x.value`.

As a concrete example, let's come up with a sequence of statements that results in `x` referring to the list (1, 2, 3). The following works fine:

```
  SchemeList x = new SchemeList();
  x.value = 1;
  x.next = new SchemeList();
  x.next.value = 2;
  x.next.next = new SchemeList();
  x.next.next.value = 3;
  x.next.next.next = null; // unnecessary since default value is already null
```

As you can see, cascading the dot operator is perfectly fine. For example, since `x.next` is a reference to a `SchemeList` object, saying `x.next.next` is legal.

The above example is very tedious to do manually, and we can make it simpler by writing functions to abstract the addition of a value into a list:

```
static void append(SchemeList thisList, int val) {
  if (thisList.next == null) {
    thisList.next = new SchemeList();
    thisList.next.value = val;
  } else {
    append(thisList.next, val);
  }
}
```

Instead of writing this function in a different module, Java allows us to combine an object type definition with a module definition. For example, if we wrote this function in the `SchemeList` object/module, the `SchemeList` definition would be:

```
class SchemeList {
  int value;
  SchemeList next;
  static void append(SchemeList thisList, int val) {
    if (thisList.next == null) {
      thisList.next = new SchemeList();
      thisList.next.value = val;
    } else {
      append(thisList.next, val);
    }
  }
}
```

Then to call `append()` on a `SchemeList x` from an outside module, we would use something like:

```
SchemeList.append(x, 2)
```

So in order to create the list (1, 2, 3), we could now do:

```
  SchemeList x = new SchemeList();
  x.value = 1;
  SchemeList.append(x, 2);
  SchemeList.append(x, 3);
```

There is still a minor annoyance in the above code segment. We had to manually set `x.value` to be `1`. In order to avoid this, we can create a *factory* function to create a `SchemeList` for us with a single value. For example:

```
static SchemeList createList(int value) {
  SchemeList thisList = new SchemeList();
  thisList.value = value;
  return thisList;
}
```

Now our code to create the list (1, 2, 3) would be:

```
  SchemeList x = SchemeList.createList(1);
  SchemeList.append(x, 2);
  SchemeList.append(x, 3);
```

By creating functions such as `createList()` and `append()`, we can abstract common operations that would be annoying to manually implement each time we wanted to do them. However, this still leaves something to desire. The code could be even less annoying if we didn't have to type in `"SchemeList"` every time we wanted to do an operation. Yes, we're getting lazy here, but laziness is the mother of all invention. Hence the inclusion of instance methods and constructors in Java.

1.3 Instance Methods and Constructors

As one more simplification to using objects, Java has *instance methods*. Such a method (or function; here I transition to using Java terminology to refer to a procedure) is associated with a particular object when it is called. The syntax for an instance method is:

```
<return type> <method name>(<type1> <var1>, ... , <typeN>, <varN>) {
  <method body>
}
```

Referring to an instance method is analagous to referring to an instance variable. The syntax is `<object reference>.<method>(<args>)`.

Besides the different syntax in calling an instance method, an instance method differs from a static method in that it has an associated object, which is bound to the variable `this`. In the call `x.method()`, `this` gets assigned the same value as `x`. So we can rewrite the `append()` method as an instance method as follows:

```
void append(int value) {
    if (this.next == null) {
      this.next = new SchemeList();
      this.next.value = val;
    } else {
      append(this.next, val);
    }
  }
}
```

Note that this method appears exactly the same as the static one we wrote before, except we replaced `thisList` with `this`. When we call `x.append(4)`, it is the equivalent of calling `SchemeList.append(x, 4)` with our `append()` methods written as above.

This is a general characteristic of instance methods. Any instance method of the form:

```
<typeR> <name>(<type1> <var1>, ... , <typeN> <varN>) {
  <body>
}
```

defined in a definition of object `<type0>` is functionally equivalent to a static method of the form:

```
static <typeR> <name>(<type0> this, <type1> <var1>, ... , <typeN> <varN>) {
  <body>
}
```

Of course, Java would not let you define the latter, since `this` is a reserved keyword, which is why we used `thisList` as a replacement above. Then the only difference between the two forms is that the first you call as `<object reference>.<name>(<arg1>, ... , <argN>)`, while the second you call as `<type0>.<name>(<object reference>, <arg1>, ... , <argN>)`.

As a concrete example, let's create the list (1, 2, 3) using instance methods. The code would be:

```
SchemeList x = SchemeList.createList(1);
x.append(2);
x.append(3);
```

Note that you cannot create an instance replacement for `createList()`, since it does not take in a `SchemeList` object as an argument but instead produces one as a result. This is an improvement over our previous code, but wouldn't it be nice if we could have a simpler replacement for the factory method?

As an alternative to factory methods, Java has special procedures called *constructors* that are executed when an object is created. Like a method, a constructor can take in arguments, but then arguments must be given when an object is created. Also like a method, we can write multiple constructors as long as they take in different arguments. The syntax for a constructor is:

```
<object type>(<type1> <var1), ... , <typeN> <varN>) {
  <constructor body>
}
```

Note that there is no return value, and the name of the constructor matches the type of the object in which it is defined. Let's write a constructor to create a `SchemeList` with a given value:

```
class SchemeList {
  int value;
  SchemeList next;
```

```
  // Constructor: creates a SchemeList with a single value
  SchemeList(int val) {
    this.value = val;
  }
  ... // other methods
}
```

This example brings up a subtle point. When the constructor is called, the newly created object is bound to the variable `this`, so we are allowed to reference it in the constructor. Indeed, how else would we assign to the instance variables of the new object from the constructor?

In order to use a constructor to create an object, the same syntax as before is used, with the possible addition of arguments. To reiterate, a constructor is used as `new <object type>(<args>)`. This creates a new object and calls the constructor corresponding to the arguments given. We can use the `SchemeList` constructor to create the list (1, 2, 3) as follows:

```
SchemeList x = new SchemeList(1);
x.append(2);
x.append(3);
```

Is this the simplest way to create our target list? That really depends on how you define simple, but we can come up with an alternative through the clever use of constructors and let the reader decide which is simpler. Let's add a constructor that also takes in a list to assign as the second member of the (value, list) pair:

```
class SchemeList {
  ... // all our other code here
  int value;
  SchemeList next;
  SchemeList(int value, SchemeList next) {
    this.value = value;
    this.next = next;
  }
}
```

Note that the variable name duplication is not a problem, since they are accessed differently. Now we can create our list with the following statement:

```
SchemeList x = new SchemeList(1, new SchemeList(2, new SchemeList(3)));
```

It's less lines of code but the tradeoff is more typing.

There is a minor issue here. Since we can write methods that take in no arguments, can we write constructors that don't take arguments? If so, how do we differentiate the expression `new <object type>()` between one that calls no constructor and one that calls a constructor that takes in no arguments? The answer is that Java does allow you to write a no-argument constructor, and that the above call always calls a no-argument constructor. In order for it to work for types that don't define a constructor, such types are provided with a default constructor that just initializes all instance variables to their default values. Note that if a type defines a one-argument constructor but not a no-argument constructor, then using `new <type>()` is illegal for that type. However, if the type defines no constructor or does define a no-argument constructor, then the expression is legal.

I'm still not satisfied by the amount of typing we had to do in our method definitions. There is lots of redundant usage of the `this` keyword, and the t, h, i, and s keys on my keyboard are really getting worn out. Can we avoid this usage of `this`? In fact, Java allows us to leave out the reference to `this`. So we can rewrite our `SchemeList` code as:

```
class SchemeList {
  int value;
  int next;
```

```
  SchemeList(int val) {
    value = val;
  }
  SchemeList(int value, SchemeList next) {
    this.value = value;
    this.next = next;
  }
  static void append(SchemeList thisList, int value) {
    if (thisList.next == null) {
      thisList.next = new SchemeList();
      thisList.next.value = val;
    } else {
      append(thisList.next, val);
    }
  }
  void append(int value) {
    if (next == null) {
      next = new SchemeList();
      next.value = val;
    } else {
      append(next, val);
    }
  }
}
```

Now the instance methods are really paying off, since the references to `this` could be removed but the references to `thisList` in the static method could not be. As was implied by our above description of `this`, a static method cannot reference `this`. Sorry.

Why didn't we remove the references to `this` in our second constructor above? In that case, a reference to `value` or `next` could mean either the parameters or the instance variables. Java defaults to accessing the parameters in this case (and it defaults to accessing local variables if their names collide with instance varaibles), so the instance variables must be accessed using `this.value` and `this.next`.

There is still one improvement we could make, though it won't really help us much in the `SchemeList`. What if multiple constructors overlap in what they do? Can we call one constructor from another? The answer is yes, though with some qualifications. There can be no cycle in constructors calling others, and the call to another constructor must be the first statement in the constructor. The syntax for calling another constructor is `this(<args>)`. As a concrete example, let's redefine our one-argument `SchemeList` constructor:

```
class SchemeList {
  ... // all our other code here
  int value;
  SchemeList next;
  SchemeList(int val) {
    this(val, null); // call two-argument constructor
  }
  SchemeList(int value, SchemeList next) {
    this.value = value;
    this.next = next;
  }
}
```

Alternatively, we could redefine our two-argument constructor to call the one-argument constructor:

```
class SchemeList {
  ... // all our other code here
```

```
  int value;
  SchemeList next;
  SchemeList(int val) {
    value = val;
  }
  SchemeList(int value, SchemeList next) {
    this(value); // call one-argument constructor
    this.next = next;
  }
}
```

Of course we can't have it both ways, since that would create a cycle. We also cannot do the following:

```
class SchemeList {
  ... // all our other code here
  int value;
  SchemeList next;
  SchemeList(int val) {
    value = val;
  }
  SchemeList(int value, SchemeList next) {
    this.next = next;
    this(value); // ERROR
  }
}
```

This is forbidden since the constructor call must be the first statement in this constructor. Also note that it is illegal to call a constructor from another method, though it is legal to call a method from a constructor from anywhere in the constructor.

1.4 Usage of Static and Instance Methods

One last question we need to answer is when to use static and instance methods. The general answer is that if a function uses the *state* (the instance variables) of an object, then it probably should be an instance method. Even more generally, instance methods should be used anywhere it makes sense to associate an object with a method. If a method is independent of any object, like our `factorial()` above, then it should be static. Another gauge to use in determining whether to use an instance or a static method is to use the one that would require the least amount of work to implement. So in the case of `append()` above, the instance method is less work than the static method, so we would use the instance method.

1.5 Conclusion

We have now done a complete introduction to functional and object oriented programming in Java. I hope this clears up any questions you had about the difference between static and instance methods and the motivations behind each of them, and when to use each.

2 Pointers

- Variables that correspond to object types don't actually hold objects, but they hold `references` or `pointers` to those objects.
- A pointer to an object is actually the location in memory at which the object is stored, but Java abstracts away this detail.
- A pointer that doesn't point to anything is a `null pointer` and is symbolized in Java by the keyword `null`. It is illegal to access fields or call methods of `null`.

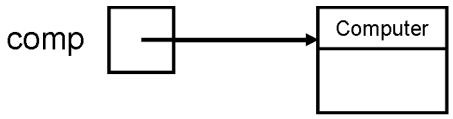

Figure 1: The value of a `Computer` variable.

3 Control Structures

- if statement:

```
if (<boolean condition>) {
  <true clause>
}
```

==OR==

```
if (<boolean condition>) {
  <true clause>
} else {
  <false clause>
}
```

If a clause is only one statement, braces surrounding it can be omitted.

Cascaded if:

```
if (...) {
  ...
} else if (...) {
  ...
} else {
  ...
}
```

- while loop:

```
while (<boolean condition>) {
  <body>
}
```

Execution starts by checking the condition. The body is evaluated if the condition is true, and the process is repeated. This continues until the condition evaluates to false, in which case execution proceeds to the next statement after the loop.

- do while loop:

```
do {
  <body>
} while (<boolean condition>);
```

This is similar to the while loop, except the condition is checked at the end of each iteration. As a result, the body will execute at least once.

Equivalent:

```
{
  <body>
}
while (<boolean condition>) {
  <body>
}
```

- for loop:

```
for (<init>; <condition>; <update>) {
  <body>
}
```

The init statement is run first, then the condition is checked. If it is true, the body is executed, the update statement is executed, and the condition is rechecked. This continues until the condition is false. Note that the init statement is only run once.

Equivalent:

```
{
  <init>
  while (<condition>) {
    <body>
    <update>
  }
}
```