Moving from CS 61A Scheme
to CS 61B Java

Introduction

Java is an object-oriented language. This document describes some of the differences between object-oriented programming in Scheme (which we hope you remember from CS 61A) and programming in Java.

An object-oriented program is a collection of agents, or *objects*, that pass information back and forth to get work accomplished. A *class* is a type of object; conversely, an object is an *instance* of a class. A class packages the data of an object with the things it can do, its *methods*. Methods are basically functions. A Java program is a collection of classes. There are no free-floating variables or functions outside of classes, as one might have in Scheme or some other language.

How this document is organized

This document starts by briefly comparing the ways Scheme and Java programs are run. Then it gives Scheme and Java versions of the bank account example program from the CS 61A course material, explaining how one would run the Java program. Next, the Scheme and Java programs are compared: first at a high level, identifying components in one program and not in the other, then in more detail, considering differences in syntax, punctuation, and so forth. The document continues with an explanation of other differences between Scheme and Java, and suggestions on how to organize a Java program. It concludes with a brief comparison of uses of recursion and iteration in the two languages.

Running a Java program

Scheme provides a “read-evaluate-print” loop that lets the programmer run a program or parts of a program interactively. To evaluate `(+ 3 4)` in Scheme, the programmer types the expression to the interpreter, which computes and prints `7`.

In contrast, one prepares a Java program with an editor, *compiles* (translates) it into a form much like machine language, then runs the result. Also, since the programming environment isn’t printing the value of each expression typed by the user, Java functions will typically print information on their own and may ask for data from the user as well. A Java program that computes the sum of 3 and 4 will contain a class; one of this class’s methods will print the result of the addition. The program must be compiled and run to produce the desired output of 7.

Scheme allows this sort of use too. For example, one can print from within a Scheme function using the `write` or `display` functions, and most Scheme programs are prepared in an editor and loaded into the interpreter using the load function. With Java, however, one doesn't have a choice.

Files that contain Java code must have names that end with the characters ".java". Typically, each file contains a single class declaration, and by convention the file name is the same as the class name with ".java" added.

An example class: a bank account

Given below are Scheme and Java versions of a bank account class declaration and use. The Scheme version appears on page 3 of "Object-Oriented Programming—Above-the-Line View" in the CS 61A course material.

Scheme

```
(define-class (account balance)

   (method (deposit amount)
      (set! balance
         (+ amount balance))
      balance)

   (method (withdraw amount)
      (if (< balance amount)
         "Not enough funds"
         (begin
            (set! balance
               (- balance amount) )
            balance) ) ) )

(define mike-account
   (instantiate account 1000))

(ask mike-account 'balance)
(ask mike-account 'deposit 100)
(ask mike-account 'withdraw 200)
```

Java

```
class Account {

   public Account (int balance) {
      myBalance = balance;
   }

   public void deposit (int amount) {
      myBalance = myBalance + amount;
   }

   public void withdraw (int amount) {
      if (myBalance < amount) {
         System.err.println ("Not enough funds");
      } else {
         myBalance = myBalance - amount;
      }
   }

   public int balance () {
      return myBalance;
   }

   public static void main (String[] args) {
      Account mike;
      mike = new Account (1000);
      System.out.println (mike.balance ());
      mike.deposit (100);
      System.out.println (mike.balance ());
      mike.withdraw (200);
      System.out.println (mike.balance ());
   }

   private int myBalance; // balance in cents
};
```

If a file named account.java contained the Java class definition above, one would compile it by typing a command that runs the Java compiler javac:

```
javac account.java
```

This would create a file named Account.class that contains the compiled version of the program. One would then run this program by typing the java command with the name of the class as argument:

```
java Account
```

The first argument to the java command is the name of a class that contains a method named main; subsequent arguments to the java command are passed to the main method.

High-level comparison

Before moving to the details of the Java code, we first compare the structure of the two versions. In the Scheme class, there are two methods, one named deposit and the other withdraw. The Java class has those two, plus three more.

- One is a *constructor* method, called to initialize an account object. It has the same name as the class. The constructor is similar to the initialize function associated with a Scheme class.
- Another is a method named main. The class whose name is given in the java command that runs the program must have a method named main whose header is as follows.

  ```
  public static void main (String[] args)
  ```

  One may pass arguments to the running program—the "args" in the method header—by typing them after the class name on the java command line.
- A third is an access method that returns the balance. Such a method is provided automatically by the Scheme object-oriented programming facility; in Java, the programmer provides it.

Not all these methods are necessary, though it is common for a class to have at least a constructor and a main method with which the class may be tested.

One other structural difference is the last thing in the Java class, namely the declaration of the variable myBalance. In Scheme, the scope of the parameter balance includes the withdraw and deposit functions, so those two functions may use and update it. In Java, functions may not be nested, so an additional variable is needed to store the object's local state. A CS 61B convention is to use names that start with "my" for local state variables.

Syntax comparison

In Scheme, essentially everything is an application of a function or a special form. These applications are combined by *composition*. Punctuation—that is, parentheses—delimit the function applications and their arguments. The syntax of Java is more complicated; a Java program includes not only function applications, but also *statements* (executed in sequence for their side effects, not their return values) and *declarations* (which convey information to the compiler). These are all punctuated differently.

Method definition

Let's look in particular at how a method is specified. Both in Scheme and in Java, a method is essentially a function. It has a header that specifies the function name and its parameters. It also has a body. In Scheme, these are all set up using parentheses:

```
(method (method-name parameter-name ...)
   expression
      ... )
```

In Java, the method header looks similar; there is no leading parenthesis, but the name precedes a parenthesized list of parameter information. The body, however, is surrounded by braces ("{" and "}"), and each statement in the body ends with a semicolon. In a method having more than one parameter, information for adjacent parameters would be separated by a comma.

```
modifiers return-type method-name (parameter-info , ...) {
   statement ;
      ...
}
```

(In general, braces are used to surround a *block*, a sequence of statements to be treated as a single unit.) The more extensive set of punctuation was incorporated in Java mainly to preserve consistency with languages such as C; however, it also allows somewhat more informative syntax error messages.

A common conventionis to start names of methods with a lower-case letter. Where a method name is a combination of two or more words, the words are run together in the name with words after the first being capitalized, e.g. removeLargest-Item.

Types

Another reason for the more complicated syntax in Java is the need to provide *type* information. In Scheme, a variable or parameter can be associated with any kind of data—list, number, symbol, or whatever. In Java, programmers must specify the type of each piece of information and of the return value of each function. The intent is to allow the compiler to detect errors resulting from assigning the wrong type of value to a variable. Four types appear in the bank account example.

type	*description*
int	Variables of type int represent integer values between -2^{31} and $+2^{31}-1$.
void	A function whose return type is void doesn't return any value at all; it is used only for its side effects.
String [ ]	String is a class provided in one of the Java class libraries. Variables of type String represent character strings. Brackets indicate a type that represents an array, an indexable sequence of elements. Here, an array of strings is used.
Account	Variables of type Account represent bank accounts.

A name is associated with a type (and the appropriate amount of memory allocated) by means of a *declaration*. Examples in the bank account code are the lines

```
Account mike;
```

in the main method and

```
private int myBalance
```

at the end of the class definition.

By common convention, names of classes (and only names of classes) start with an upper-case letter. Where a class name is a combination of two or more words, the words are run together in the name with all the words being capitalized, e.g. CheckingAccount.

Method calling

A third syntactic difference between Scheme and Java is the different notation for calling a method. In Scheme, it's

```
(ask object method-name argument ...)
```

In Java, it's

```
object.method-name (argument ...)
```

that is, the object name followed by a dot followed by the method name. Thus the Scheme call

```
(ask mike-account 'deposit 100)
```

would be coded in Java as

```
mike.deposit (100);
```

Occasionally a method will call one of the object's own methods. In Scheme, this is done with the expression

```
(ask self method-name ...)
```

The Java counterpart for self is named this. Thus an atmDeposit method that results in a $.50 usage fee might be coded as

```
public void atmDeposit (int amount) {
   myBalance = myBalance + amount;
   this.withdraw (50);
}
```

The this object need not be specified in such a situation, however, so the method call may be written as

```
withdraw (50);
```

Some functions in Java are represented by *infix* operators that appear between their operands, as in algebraic expressions. Examples include

the + and – arithmetic operators, named the same as in Scheme;
the < and > comparison operators, also named the same as their Scheme counterparts;
the == comparison operator, the counterpart of = in Scheme;
the && and || logical operators, the counterparts of and and or in Scheme; and
the = assignment operator, similar to set! in Scheme.

The parentheses in Scheme expressions determine the order in which functions are applied. In Java, however, one may write complicated expressions without parentheses, for example,

```
a + b > 27 && b + c < 35 || a < 3
```

Possible ambiguities are resolved by assigning each operator a *precedence* that specifies how much "stronger" it is than other operators. The above expression, fully parenthesized, would be

```
((((a + b) > 27) && ((b + c) < 35)) || (a < 3))
```

Comments

Comments in Scheme start with a semicolon. There are two kinds of comments in Java: one-line comments that start with // (two slashes), and multi-line comments that start with /* (slash-star) and end with */ (star-slash).

Case sensitivity

Unlike in Scheme, names in Java are case-sensitive; thus the names MIKE and mike are different. Java names may contain only letters, digits, and the characters "$" and "_" (underscore), and may not start with a digit.

Typical conventions for use of upper- and lower-case letters in Java programs are as follows. Class names are capitalized, with words within the name also starting with a capital letter (example: BankAccount). Names of *constants*—variables whose values don't change during the execution of a program—are often typed with all upper-case letters (example: MAXLENGTH). Other identifiers start with a lower-case letter, with upper-case letters used to begin each word within the identifier (example: removeLargestItem).

Other aspects of values, objects, and methods

Primitive types and references

Java's int type is a *primitive* type, since an int value is not composed of any other values. The other primitive types are bool—a type for representing true or false values—and several other types for representing numbers. Primitive values are Java's counterpart to Scheme atoms.

In Scheme, lists are represented by pointers. Similarly in Java, objects and collections of data are also represented by pointers, referred to as *references*; the corresponding types are *reference types*. Java arguments are passed to methods *by value*, in exactly the same way as arguments are passed to Scheme functions. Thus the effect of changing an argument of a primitive type within a method is confined to the method itself, while a change to an object or collection whose reference is passed as an argument will stay changed when the method is exited.

Object instantiation

An object is instantiated in Scheme with the instantiate function and in Java with the new operator. In the bank account example, the expression

```
(define mike-account (instantiate account 1000))
```

did double duty; it added the name mike-account to the relevant environment and associated it with the account object. The Java counterpart is

```
Account mike = new Account (1000);
```

or, split into its component parts,

```
Account mike;
mike = new Account (1000);
```

The new operator returns a reference to the constructed object.

Return values

The value returned by a Scheme function is specified implicitly; it's the value of the last expression in the body. The value returned by a Java method is specified explicitly by using the return statement, as in the balance method in the Account class. Thus every function whose type is not void must contain at least one return statement with an argument expression of the correct type. The return statement without an argument is used in void functions; an implicit return appears at the end of each void function.

Names and accessibility

The keywords public and private specify who can call a method or reference a variable. A public method may be called from a method in any other class; a private method may be called only from methods of the same class. Methods defined using the method special form in Scheme are all public; to get the effect of a private method in Scheme, one would merely define it as a regular function internal to the class definition.

The scope of a variable declaration in Java is the rest of the block that contains the declaration (that is, up to its closing brace).

In Scheme, there is a single "name space" for functions and variables. For example, the definition of a variable named list makes the builtin function list inaccessible, and the definition of one function named foo replaces any earlier definition of that function. In Java, however, function names are separate from variable names, so you may have a method and a variable with the same name.

Also, method names may be *overloaded*—that is, there may be two different methods with the same name. They must, however, have a different number of parameters or parameters of different types. An example where overloading might be useful is a class that represents a length expressed in feet and inches. One might code two versions of this class's "add" method, one taking one argument representing some number of inches, the other taking two arguments that represent feet and inches. (One may not have two methods that have a single integer parameter, say, one having an "inch" parameter and the other having a "foot" parameter. The reason is that if you try to call add(3), the Java compiler wouldn't know which one to use.)

Methods and variables declared as static are common to all the objects of a class. (Static variables are called "class variables" in the CS 61A course material.) A method (e.g. main) must be declared as static if it's called before any objects of the corresponding class have been instantiated.

Packages of classes

A Java programming environment provides numerous *packages* of predefined classes and objects. A programmer might refer to one of these classes by its fully qualified name, for example, java.math.BigInteger, or use the import statement in order to use only the class name:

```
import java.math.*;
```

How to write a Java program

Moving from Scheme to Java involves mastering a lot of detail, some of which has been described above. One way of organizing this detail is to put it into the context of building a Java program. Here are some suggestions.

1. Figure out what objects will be involved in the program, and how they will interact. (Some objects will model agents or things in the real world, like bank accounts, workers, teaching assistants, or singers; others implement programming services, such as a collection of data, a source of input, or a provider of computational routines.) It may help for you and a friend to *act out* the interactions, with one of you playing one object and one of you playing another.

   One aid to design is to recognize object patterns. One such pattern that appears often in CS 61B is that of a *container*. Its methods include the following:

   initialize;
   insert an element;
   remove an element;
   determine if a given value is an element;
   the main method (for testing).

2. Create a framework for each class (in a file of its own). The framework starts with the line:

   ```
   class class-name {
   ```

   It ends with a single right brace:

   ```
   }
   ```

   In between are frameworks for the methods:

   ```
   modifiers return-type method-name ( argument-info ) {
   }
   ```

   One of the methods is main. (It's not absolutely necessary but is useful for testing the class methods.)

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

   For example, a framework for the container class mentioned above would be

   ```
   class ContainerFor61B {
      public ContainerFor61B () {
      }
      public void Insert (element e) {
      }
      public void Remove (element e) {
      }
   ```

```
      public bool Contains (element e) {
      }

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

Most methods are public; most data variables are private. Exceptions are private methods that are helper functions to some other public methods, but are not needed outside the class.

3. Fill in the method bodies and add private data.

Recursion vs. iteration

Programming in CS 61A involved a lot of recursive programming; three recursive versions of the factorial function from Abelson and Sussman appear below.

Section 1.2.1

```
(define (factorial1 n)
   (if (= n 1) 1 (* n (factorial1 (- n 1))) )
```

Section 1.2.1

```
(define (factorial2 n)
   (define (iter product counter)
      (if (> counter n) product
         (iter (* counter product) (+ counter 1))) )
   (iter 1 1) )
```

Section 3.1.3

```
(define (factorial3 n)
   (let ((product 1) (counter 1))
      (define (iter)
         (if (> counter n) product
            (begin
               (set! product (* counter product))
               (set! counter (+ counter 1))
               (iter) ) ) )
      (iter) ) )
```

Given below are the Java counterparts.

```
class Example {

   public static int factorial1 (int n) {
      if (n == 1) {
         return 1;
      } else {
         return n * factorial1 (n-1);
      }
   }

   public static int factorial2 (int n) {
      return iter (1, 1, n);
```

```
        }

        private static int iter (int product, int counter, int n) {
           if (counter > n) {
              return product;
           } else {
              return iter (counter*product, counter+1, n);
           }
        }

        public static int factorial3 (int n) {
           int product=1;
           for (int counter=1; counter<=n; counter=counter+1) {
              product = counter * product;
           }
           return product;
        }

        ...
     }
```

The imperative style of coding—represented by factorial3—is more common in Java. The examples above, however, show that recursion may also be easily used in Java.