Structure of Java Programs

Below is a simple description of what can go into a Java program and the syntax that you have to follow. This isn't a complete syntax of the Java language. However, it does cover the things that we are likely to use during the semester. Things that are in italics are defined as one of the elements in the syntax. I have used square brackets to denote things that are optional. The exception to this will come with arrays where square brackets are used in the Java syntax. I use the pipe, |, to represent or if something can be one of two or more options.

Classes:
public class *Name*[<*GenericInfo*>] [extends *ClassType*] [implements *InterfaceType1* [,*InterfaceType2* ,...]] {
    [*Constructors*]
    [*Methods*]
    [*MemberData*]
    [*InnerClass*]
    [*Enumerations*]
}

This defines a class with the specified name that can inherit from a specified class and might implement one or more interfaces. All the methods and member data for this class go inside of it. We probably won't write any classes this semester that use the generic information either. Inner classes and enumerations will only pop up much later.

Constructor:
public ClassN*ame*([*Type1 name1*[,*Type2 name2*, ...]]) {
    [*Statements*]
}

Constructors are methods that are called when an object is created. They have to have the same name as the class they are in.

Method:
public [static] [final] *Type name*([*Type1 name1*[,*Type2 name2*, ...]]) {
    [*Statements*]
}

This defines a method with the given name. The argument list is in the parentheses. It is a comma separated list with zero or more type name pairs. Not all methods have to be public. They can be private or protected but you will likely make them public by default. Final isn't likely to be used for methods frequently in this class.

Member Data:
private [static] [final] *Type name*[=*expression*];

This defines a class level variable with the specified name and type. The initial value doesn't have to be specified. Not all data members have to be private, but they should be unless they are static and final. Final means that once the variable is given a value, it can't be changed.

Type:
PrimitiveType | *ObjectType*

PrimitiveType:
boolean | char | byte | short | int | long | float | double | void

While all of these are valid primitive types, this class is unlikely to use anything but boolean, char, int, and double. The type void can only be used as the return type of a method, not for a variable declaration.

ObjectType:
ClassType[<*GenericInfo*>] | *InterfaceType*[<*GenericInfo*>]

ClassType:
This is the name of a class. It can be one that you have written or one that is in an library. Class names typically start with a capital letter and have each new word capitalized. They follow the normal rules for Java names.

InterfaceType:
This is the name of an interface. It can be one that you have written or one that is in an library. Interface names typically start with a capital letter and have each new word capitalized. They follow the normal rules for Java names.

Statement:
VariableDeclaration | *Assignment* | *MethodCall* | *ReturnStatement* | *CodeBlock* | *IfStatement* | *ForLoop* | *WhileLoop* | *SwitchStatement* | *DoLoop* | *LocalClass*

VariableDeclaration:
[final] *Type name*[=*expression*];

Assignment:
name=expression;

This stores the value of an expression in the variable with the given name. The variable must be in scope. So it either has to be declared in the current method above this line or it should be a class level variable. In order for this to work, the types need to match or the expression type must be safely convertible to the variable type.

There are also special operators that do assignment. These include increment (++) and decrement (--). In addition, any binary operator can be combined with an = to make an assignment operator. For example, i++ is the same as i=i+1. Similarly, i+=8 is the same as i=i+8.

(Technically all assignments are expressions that return the value that is assigned. Any such expression followed by a semicolon is also a valid Java statement.)

MethodCall:
objectExpression.methodName([*expression1* [, *expression2*, ...]]);

The objectExpression is any expression whose type is an object type. The methodName must be a

valid name of a method on that type. The expression is often just the name of a variable, but it can itself be a method call that returns an object.

ReturnStatement:
return [*expression*];

This statement stops the execution of a method and returns the value of the specified expression. If the method has a void return type, then there isn't an expression. Otherwise the there must be an expression and the expression type must match the return type of the method or be convertible to it.

CodeBlock:
{ [statements] }

Any place a single statement is expected in Java, it can be replaced by multiple statements inside of curly braces. They don't all have to be on the same line as I have typed it here.

IfStatement:
if(*booleanExpression*) *statement1* [else *statement2*]

This is the most basic conditional statement. If the boolean expression is true, statement1 will execute. If it is false either nothing happens of statement2 is executed if the else clause is used. A common usage might look like the following:

```
if(a==5) {
        b=25;
}
```

ForLoop:
for(*statement1*; *booleanExpression*; *statement2*) *statement3*

This is the most commonly use loop structure in Java. When it starts, statement1 is executed. It is often called the initializer because it typically sets things up and often declares a variable. It will then repeatedly evaluate the boolean expression and if it is true it executes statement3 followed by statement2. Statement3 is called the body of the loop and statement2 is the iterator. A common usage might look like the following.

```
for(int i=0; i<10; ++i) {
        System.out.println(i);
}
```

Here we declare a variable called i that is an integer and start it off at zero. We keep going as long as i<10. Each time we print the value of i and then increment it.

WhileLoop:
while(*booleanExpression*) *statement*

This loop evaluates the expression and if it is true it executes the statement. This is repeated over and over until the expression evaluates to false.

SwitchStatement:
switch(*intExpression*) {
case 1:
    [*statements*]
    break;
case 2:
    [*statements*]
    break;
case 3:
    [*statements*]
    break;
...
default:
}

The switch statement allows you to branch to one of many options based on the value of an integer expression. The values don't have to start at 1 as I have shown here. They can be whatever you want. Break statements are also optional, but they are almost always needed. Break statements can also be used in loops to terminate the loop, but I won't ever do that.

DoLoop:
do {
    [*statements*]
} while(*booleanExpression*);

This is like the while loop except that the statements happen at least once before the expression is evaluated.

LocalClass:
You can declare classes inside of methods. They have basically the same rules as the overall class.

Expression:
An expression in Java is anything that has a type and value. Below is a list of options.

- Variable name – The name of a variable evaluates to the value currently stored in that variable.
- Numeric literal – Numbers are expressions. You can represent integer values or decimal fractions as you normally would. Very large or very small numbers are written using scientific notation of the form 1.234e56 which is the number $1.234*10^{56}$, a rather large number indeed.
- String literal – String literals can be any sequence of characters enclosed in double quotes. For example, “Hi mom.” is a string literal.
- Method call – A method call for any method that returns something other than void is an expression.
- Type casts – You can force Java to do type conversions that might lose information or not work by putting the type you want in parentheses in front of an expression. So if number is a variable of type double, (int)number will return only the whole number part as an int type. It can be done with object types in what is called a narrowing cast.
- Operator expression – Java supports different mathematical and logical operators. Most will join two different expressions. Some operate on one expression and one uses three expressions. This is a list that includes most of the ones we are likely to use.

- A+B : Standard addition for numeric values. If either A or B is a String the plus sign will do string concatenation.
- A-B : Subtraction for numeric values.
- A*B : Multiplication for numeric values.
- A/B : Division for numeric values.
- A%B : Modulo operator that returns the remainder after division for integer values.
- A==B : This returns true if A is the same as B. For object types use .equals instead.
- A<B : Returns true if A is less than B. Works for numeric types.
- A>B : Returns true if A is greater than B. Works for numeric types.
- A<=B : Returns true if A is less or equal to than B. Works for numeric types.
- A>=B : Returns true if A is greater than or equal to B. Works for numeric types.
- A!=B : Returns true if A not equal to B.
- A[B] : Requires A to be an expression of an array type and B to be an expression of an integer type. This has the value of the Bth element of A.
- A || B : This boolean operator is true if A or B is true. It is inclusive or so it is true if both are true.
- A && B : The boolean operator returns true only if both A and B are true.
- !A : This boolean operator is the not operator. It returns true if A is false and false if A is true.
- A?B:C : This is the ternary operator. A should be a boolean. If A is true, the value will be B, otherwise the value will be C.
- A instanceof C : Here A is an expression with an object type and C is the name of a class. It tell you if A is of the type C.

Names:
Valid Java names have to start with a letter or underscore followed by any number of letters, underscores, or numbers.

InnerClass:
You can declare classes inside of classes. They can be public or private. They can also be static. I won't go into the details of those here.

Enumeration:
enum *Name* {*Name1*, *Name2*, ...};

Enumerations allow you to define a type that can take on one of a small set of values. For example, a street light can only be red, yellow, or green. Nothing else should be allowed. An enumeration allows you to do this. We won't go into any detail on how to use these this semester.

GenericInfo:
ObjectType1 [, *ObjectType2*,...]

This is just a comma separated list of object types that are the generic information for a class or interface.